University of Ulm | 89069 Ulm | Germany

Faculty of Engineering and Computer Science
Department of Databases and Information Systems

Hand Gesture-based Process Modeling for Updatable Processes

Bachelor Thesis at the University of Ulm

Submitted by:
Hayato Hess
hayato.hess@uni-ulm.de

Reviewer:
Prof. Dr. Manfred Reichert

Supervisor:
Jens Kolb

2013

Version July 30, 2013

Satz: PDF-LaTeX 2_ε

Abstract

The increasing popularity of process models leads to the need of alternative interaction methods to view and manipulate process models. One big research field are the gesture-based manipulation methods. Although there are already works in this research area [KRR12, Dap12], they utilize only two dimensions for gesture recognition. The objective of this work is to introduce a system that manipulates process models using a three dimensional hand gesture input interface utilizing the RGB-D camera of the *Microsoft Kinect*. With this, an input interface can be created that is more natural and, thus, is easier to learn and use than its two dimensional counterpart.

This work therefore discusses how gestures are recognized as well as technical implementation aspects (e.g., how process models are painted, accessed and manipulated). Furthermore, it explains the problems arising from the use of the Kinect as a hand tracking system and shows which steps have been taken to solve these problems.

Acknowledgement

First and foremost, I thank my supervisor Jens Kolb for his invaluable guidance. Second, I thank my parents and my friends for the patience and the constant support. Third, I want to thank Prof. Dr. Manfred Reichert who showed great interest in this work and agreed to be the reviewer for this thesis. Forth, I thank the department of *Database and Information Systems* for providing the hardware and support for this work. Last, I thank Nintendo who's games delighted me in the darkest hours of writing.

Contents

1 Introduction

The idea of creating gesture-based user input is not new. The movie *Minority Report* (2002) is still well remembered because of the alternative input methods for information systems. There are already various types of gesture-based input systems available. Therefore, creating a gesture-based information system one has to ask whether similar systems already exists and if the usability is appropriate for the specific application domain. Gesture recognition to manipulate process models based on multi-touch tablets already exist [KRR12]. However, this system utilizes only two dimensions. Systems utilizing the third dimension for gesture-based process models manipulation do not exist yet. Therefore, the decision was made to create a system capable of gesture-based process model manipulation utilizing all axis of the three dimensional space.

The usability perspective may be discussed controversially. On the one hand, a user that uses a computer with the default input is presumably faster than one that has to rely on gestures. On the other hand, however, there are cases where a traditional input method proved to be inefficient. For example, when the user needs one of his hands for another activities like holding the steering wheel or has a handicap and is thereby unable to use a mouse or a keyboard. Another point is, that more space is required for traditional input methods. A computer with a mouse and a keyboard requires more space than a camera that can be mounted almost everywhere. Furthermore, the input with three dimensional gestures can be made very intuitive. This is important for people who have troubles working with computers or understanding a process work flow. With the three dimensional gesture-based system, they can interact with easy to learn gestures. Therefore the vocational adjustment time would be lower for them than with traditional input methods where they might get lost. It is also more natural to interact by using the hands as an input method. Humans tend to use their hands in a conversation. Why not tell a system our needs in the same way?

This work is divided into five sections. First, Section 2, covers the basics for further sections. Second, Section 3, introducing a library with its components to communicate with a remote server to obtain process models. Third, Section 4, explaining an algorithm used to paint process models on the screen. Forth, Section 5, introducing the gesture recognition algorithm and explaining problems and their solutions when working with a Kinect as gesture recorder. Last, Section 6, showcases a three dimensional gesture-based process model manipulation prototype.

2

Background

This section covers theoretical concepts that serve as basis for the next sections. In Section 2.1, process models are covered. Section 2.2 explains a framework used to access and manipulate Process Models. Section 2.3 covers an algorithm used for gesture recognition.

2.1 Process Model

A *process model* is a directed graph that consists of a set of *nodes* N and *edges* E (cf. Figure 2.1).

As described in [KKR12b, KR13a], the nodes N can be divided into:

- *StartFlow*, *EndFlow*; Thereby, the StartFlow node represents the start of the process and EndFlow its end.
- *Activity* nodes having exactly one incoming and one outgoing edge and representing tasks in a process model.
- *Gateway* nodes, like *ANDsplit*, *ANDjoin*, *XORsplit*, *XORjoin*, *LOOPsplit*, and *LOOPjoin*, have multiple incoming and outgoing edges and may conditionally control the process flow. ANDsplit and Andjoin split the current path into two parallel executed paths. XORsplit and XORjoin do the same however with the restriction that only one of the two paths is executed conditionally. LOOPsplit and LOOPjoin gateways are used to conditionally repeat the paths between them.
- *Data Elements* represent data storages (e.g., variables or an external database). They are accessed or written by activity nodes.

The edges E can be divided into [KKR12b, KR13a]:

- *ET_Control* edges, used to show the execution order of activities and gateway nodes.
- *ET_SoftSync* edges, used for synchronization of parallel executed paths.
- *ET_Loop* edges, used to repeat the paths between the LOOPjoin and the LOOP-split node.
- *ET_DataFlow* edges connect data elements and activities. The direction of the edge shows whether the data element is accessed to read or write.

Definition 1 (Relative Component Operator)**.** *Let $P = (N, E)$ and $P' = (N', E')$ be process models with the nodes N, N' and edges E, E'. Then $P \backslash P' = (\{n \in N | n \notin N'\}, \{e \in E | e \notin E'\})$.*

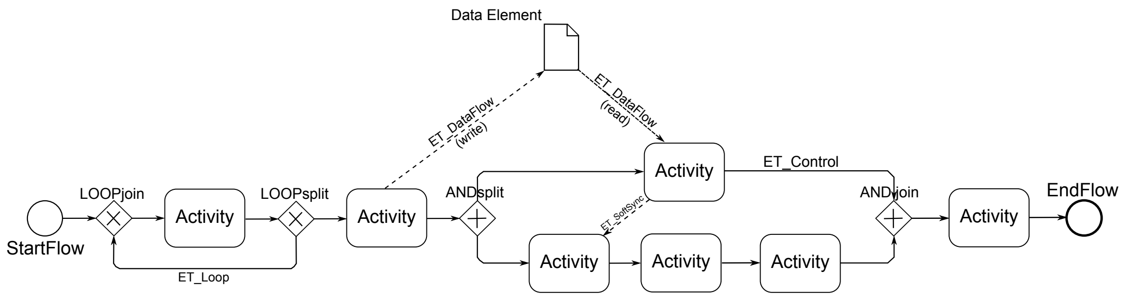


Figure 2.1: Example of a Process Model

Definition 2 (Single Entry Single Exit Blocks)**.** *The process model $P = (N, E)$ consists of subgraphs. These subgraphs $S = (N', E')$ with $N' \subset N, E' \subset E$ are called SESE (Single Entry Single Exit) blocks, iff there is exactly one incoming and one outgoing ET_Control edge that connects S with $P \backslash S$ [KR13a].*

2.1.1 Central Process Model

A *Central Process Model* (CPM) is a *process model* as defined in Section 2.1. It represents the information of the corresponding business process and serves as a basis to create respective process views. Furthermore, a CPM may contain program logic required to execute it within a process-aware information system.

2.1.2 Process View

A process view is an abstraction of a specific CPM for specific needs and is created by applying a sequence of *view create* operations $\langle Op_1 \dots Op_n \rangle$ with $Op_i \in \{aggregation, reduction\}$ to a CPM. *Aggregation* merges several nodes to one abstracted node and requires a SESE block to work on. The *reduction* is used to remove a set of nodes from the process view (cf. Section 2.1) [KR13a].

Note that all view create operations Op_i used to create a process view do not affect the underlying CPM.

Modifying the process view and CPM, so-called *update operations* may be applied. Update operations are operations that insert or delete nodes or edges in the process view and its respective CPM.

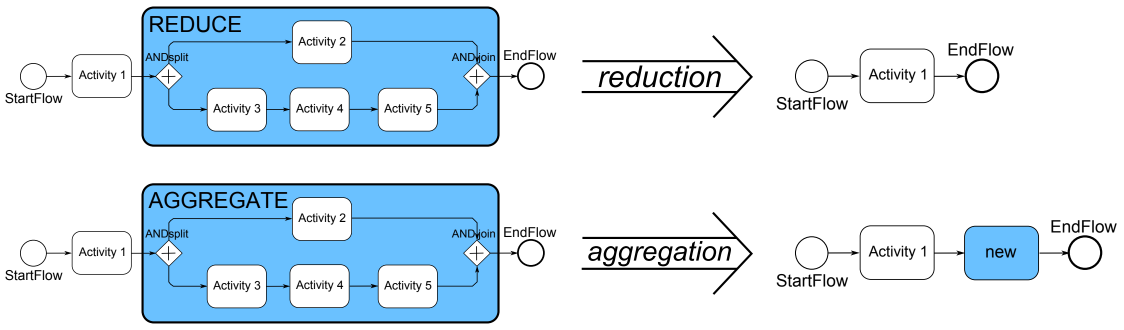


Figure 2.2: Reduction and Aggregation Operation

2.2 The proView Framework

The *proView* framework is based on the client-server model and, therefore, consists of a client and a server component. The *proViewClient* is responsible for visualizing the process models and handling user input. The *proViewServer* is responsible for monitoring and performing changes to the stored process models. Furthermore, it provides a *Representational State Transfer* (*REST*)[1] interface that enables the client to retrieve, modify, add, and delete stored models.

The *proView* framework already has an implementation as described in [KKR12a, KKR12b, KR13b]. The proViewServer, written in *JAVA*, is reused. The proViewClient, written in *Vaadin*[2], is replaced by a newly created client, written in C**#** (cf. Section 6.1). This new client interacts with the already implemented proViewServer and handles process graph visualization and Kinect-based user input.

[1]Representational State Transfer – a software architecture that can be used for distributed systems in the world wide web.
[2]Vaadin – a web-framework for building modern web applications.

2.3 Dynamic Time Warping

Dynamic Time Warping (*DTW*) is a dynamic programming algorithm used for speech recognition [MR81], because of its ability to cope with different speaking speeds and pauses. Requirements for gesture recognition are similar since different users perform the same gesture with different speeds. Therefore, DTW is able to match two input gesture data streams (e.g., of a camera) that may have different lengths and calculate their similarity; the so-called *DTW distance* [LWZ$^+$09]. As a distance function, the *Euclidean distance* is applied in an one dimensional space (cf. Figure 2.3). More precisely, the function calculates the *local distance* between two points of the input data streams. The DTW distance is the minimum of the sum of all local distances of each possible path.

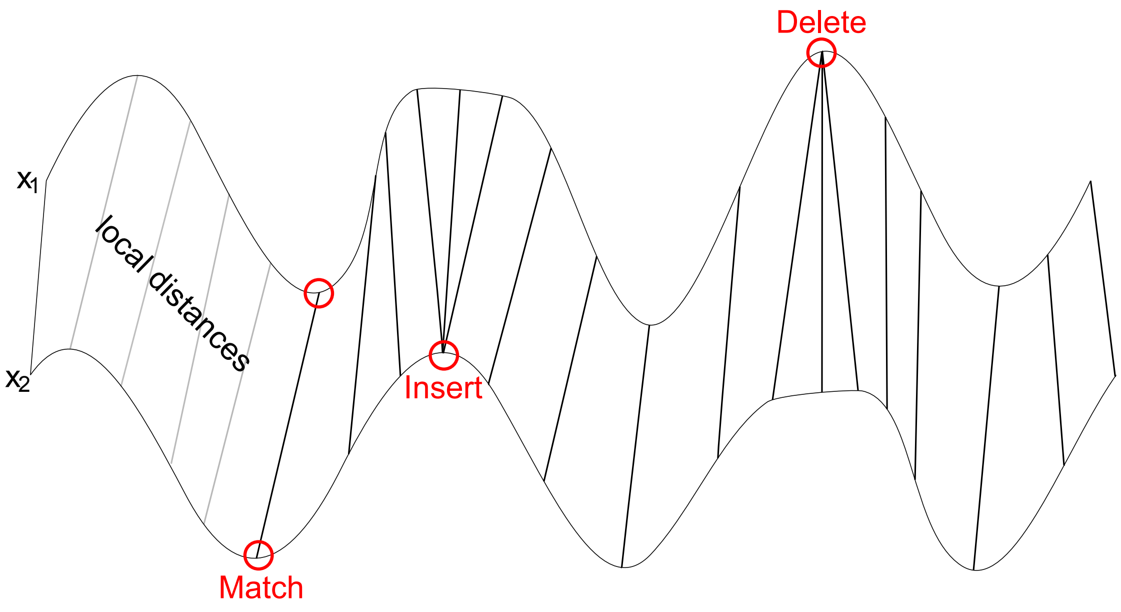


Figure 2.3: DTW Match of x_1 and x_2.

```
1  double dtw($x_1$ [1..n], $x_2$ [1..m]) {
2      // initialize Table
3      Table := [0..n, 0..m]
4      Table[0, 0] := 0
5
6      for i := 1 to n
7          Table[i, 0] := infinity
8      for i := 1 to m
9          Table[0, i] := infinity
10
11     // calculate DTW using the table
12     for i := 1 to n
13       for j := 1 to m
14         cost:= distance($x_1$[i], $x_2$[j])
15         Table[i, j] := cost + minimum(
16                                Table[i-1, j  ], // insert rule
17                                Table[i  , j-1], // delete rule
18                                Table[i-1, j-1]  // match rule
19                              )
20     return Table[n, m]
21 }
```

Listing 2.1: DTW Algorithm

The DTW calculation is done as follows (cf. Listing 2.1): It first initiates a table with the dimensions of the lengths of the input data streams (lines 3-4) and then calculates the DTW distance by using a distance, a minimum function, and three rules. First of all, the *insert* rule (line 16) that matches multiple points in the data stream x_1 to one point in x_2. Second, counterpart to insert, the *delete* rule (line 17) used to match multiple points in x_2 to one in x_1. Finally, *match* rule (line 18) matches one point of x_1 to one of x_2. After calculating all values in the table, the DTW distance is determined by the entry in the last row and column (line 20). The paths leading to the *optimal distance*, the DTW distance, may be calculated via backtracking, if required. These paths represent the points that were matched and, therefore, contain the information about the similarities of the two data series.

In the use case of the gesture recognition, a gesture can be recognized by comparing it to a recorded gesture and calculating the DTW distance. If the gestures are very similar, the DTW distance has a low value via versa. A gesture is therefore recognized, when the DTW distance falls below a threshold.

3 REST Library

This section covers the basic structure of the «REST Library» and sheds light on how the communication with the proViewServer is done. Section 3.2 explains the communication protocol. Section 3.2 deals with getting process model updates from the proViewServer. Section 3.4 explains how process model changes are handled. Section 3.5 showcases a communication example between «REST Library» and proViewServer.

3.1 Interface

The intention of the «REST Library» is to encapsules the communication with the proViewServer and provide the user a list of Central Process Models (CPM), which is represented through a *CPM class* (cf. Figure 3.1). Remember, each CPM represents a whole business process and is the basis to create personalized process views. For that reason CPM classes reference *process view classes*. Both, the CPM and the View classes extends the *model class*. The model class contains elements required to represent a process model. The View classes provides all operations (e.g., AggregateNodes, ReduceNodes, DeleteNodes, InsertSerialNode, InsertParallelNode, and InsertSyncEdge) to manipulate the View and their underlying CPM. Generally, users of the «REST Library» can handle processes as if they are stored locally; although they are stored remotely on the proViewServer.

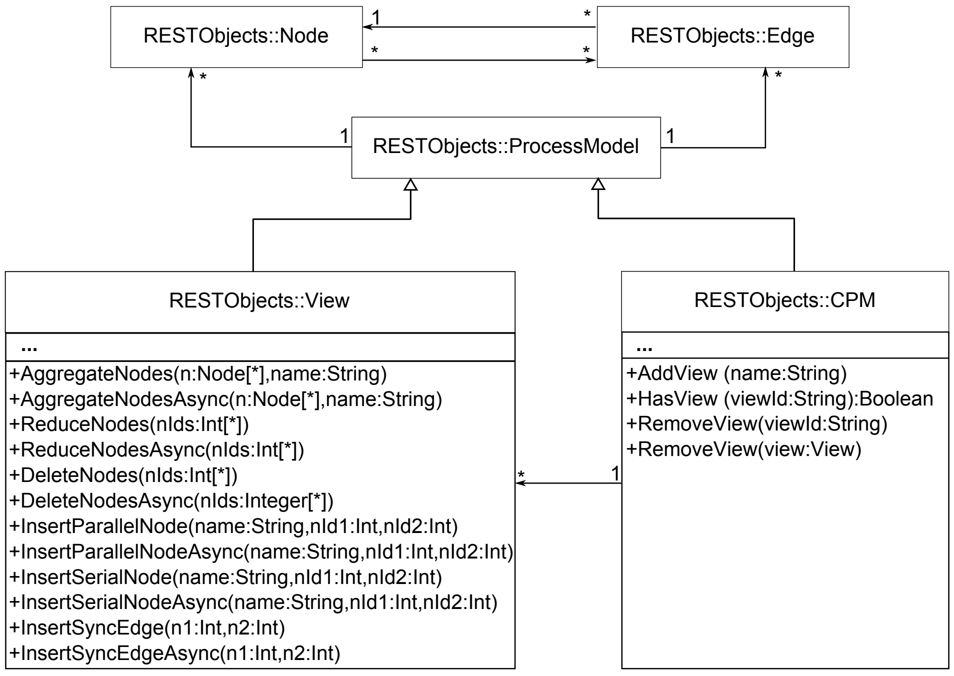

RESTObjects::Node
RESTObjects::Edge
RESTObjects::ProcessModel
RESTObjects::View
...
+AggregateNodes(n:Node[*],name:String)
+AggregateNodesAsync(n:Node[*],name:String)
+ReduceNodes(nIds:Int[*])
+ReduceNodesAsync(nIds:Int[*])
+DeleteNodes(nIds:Int[*])
+DeleteNodesAsync(nIds:Integer[*])
+InsertParallelNode(name:String,nId1:Int,nId2:Int)
+InsertParallelNodeAsync(name:String,nId1:Int,nId2:Int)
+InsertSerialNode(name:String,nId1:Int,nId2:Int)
+InsertSerialNodeAsync(name:String,nId1:Int,nId2:Int)
+InsertSyncEdge(n1:Int,n2:Int)
+InsertSyncEdgeAsync(n1:Int,n2:Int)
RESTObjects::CPM
...
+AddView (name:String)
+HasView (viewId:String):Boolean
+RemoveView(viewId:String)
+RemoveView(view:View)
1
*
1
1
*

Figure 3.1: «REST Library» Components.

3.2 Communication Protocol

In the following, the communication with the proViewServer using *REST* messages is explained. At first the communication is illustrated by an example then the different types of requests and replies are explained.

3.2.1 Communication Example

The UML sequence diagram in Figure 3.2 visualizes a communication sequence between the proViewServer and the «REST Library». At first, the «REST Library» needs to get a process view or CPM to display. Therefore, it requests the *process list* (cf. Section 3.2.2) from the proViewServer. The proViewServer replies with an XML file containing a list of CPMs and their process views (cf. Section 3.2.2). Then the «REST Library» selects a specific process view by its ID and sends a GET request containing the respective ID (cf. Section 3.2.3). The proViewServer replies to the request with the requested process model (cf. Section 3.2.4).

The process model may be used to be displayed to the user. In our example, the user deletes activity *GetDocument* with the ID "5". Therefore, the «REST Library» sends an UPDATE activity request (cf. Section 3.2.3) providing the ID of the activity. In Figure 3.2, this request is successful and the server returns an XML that states this fact (cf. Section 3.2.4). Due to the successful delete request, the process view and the respective CPM are modified. Therefore, the «REST Library» sends GET requests to obtain the updated versions of the process models (cf. Section 3.2.3). The resulting process models are parsed by the «REST Library» and its observers are notified about the change.

Note that the requests in this example are all synchronous. The «REST Library» supports asynchronous requests as well (cf. Figure 3.1).

3.2.2 Acquire List of Process Models

To get a list of available process models, one may simply invoke the address `<ip of the proview server>:8182/process?list` of the proViewServer using a HTTP GET request. The address varies depending on the configuration in the *ProView.properties* settings file of the proViewServer.

The sequence diagram in Figure 3.3 demonstrates how the list is retrieved by using a synchronous request.

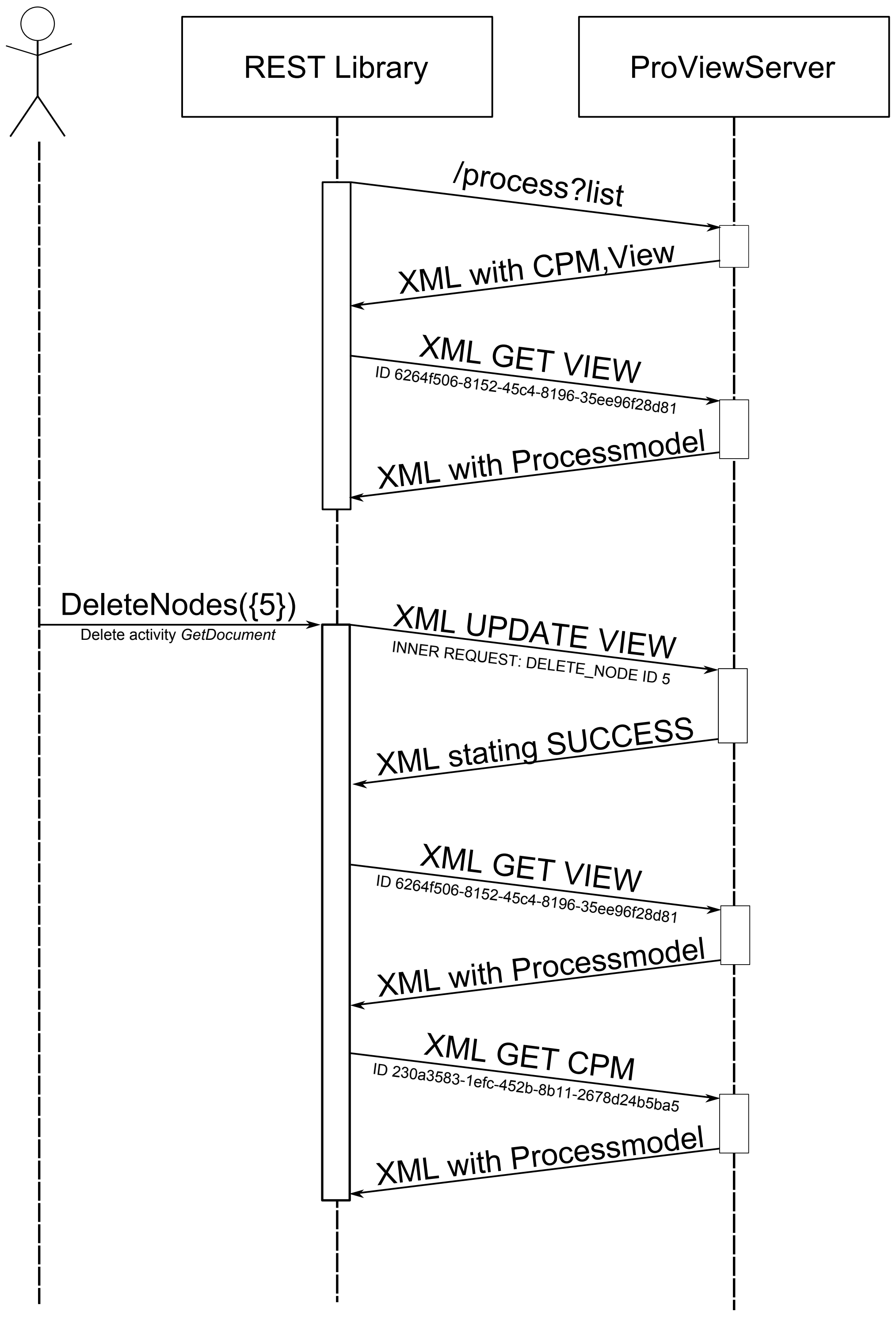


Figure 3.2: Sample Communication.

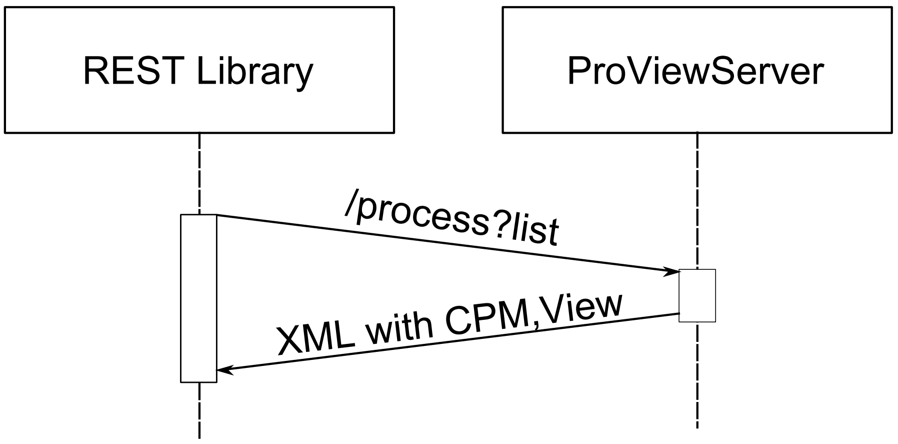


Figure 3.3: Acquire List of Process Models.

Response The proViewServer replies to the HTTP GET request with an XML file (cf. Listing 3.1) containing all available process models (i.e., CPMs and process views). Therefore, the XML root element `<models>` contains a set of CPM. Each CPM has an *ID* (lines 3,7), *name* (lines 4,8), and *version number* (lines 5,9). Furthermore a CPM model may comprise multiple process view models (lines 10-13). Each process view has its own *ID*, *name*, and *version* (lines 10-12).

```
1  <?xml version="1.0" ?>
2  <models>
3    <cpm cpmID="0bf59cff-801f-4249-896b-4b9f8c50ac9a"
4         cpmName="CreditApplication"
5         cpmVersion="1"
6    />
     ...
7    <cpm cpmID="17d84280-adb3-4083-bb63-9bd9b525a94c"
8         cpmName="Complex Process"
9         cpmVersion="10">
10        <view viewID="3d548cc0-5413-41a5-928b-54b0f292d405"
11              viewName="first view"
12              viewVersion="15"/>
          ...
13   </cpm>
     ...
14 </models>
```

Listing 3.1: Response Containing CPM and Process Views

3.2.3 Request a Change on a Process View or CPM

Requesting a process model based change on the list of CPM and corresponding process view, a HTTP POST request has to be sent to `<ip of the proview server>:8182/view/request` (cf. Figure 3.4). Thereby, the XML payload describes the type of request and what has to be changed. The XML payload is divided into two parts.

Part 1: As shown in Listing 3.2, the first part (lines 2-6) describes what operation is requested (line 3) and which type of resource (i.e., which `CPM` or `VIEW`) is addressed (line 6).

```
1 <?xml version="1.0" ?>
2 <Request>
3   <requestOperation>CREATE|DELETE|GET|UPDATE</requestOperation>
4   <resourceID>ID</resourceID>
5   <payloadFormat>XML</payloadFormat>
6   <resourceType>(CPM|VIEW)</resourceType>
7  [<query> ( viewName=name )|( Inner Request ) </query>]
8 </Request>
```

Listing 3.2: Update and Create Process View Requests

`GET` request operations is sent to the proViewServer to get the content of a process view or CPM (i.e., the actual process model). Therefore, the *resourceID* of the respective process model has to be given.

`CREATE` and `DELETE` request operations are applied to add and remove process views from the proViewServer. Both the `CREATE` and `DELETE` operation require a `resourceID` (line 4). The `resourceID` in the `DELETE` operation is used to identify the specific process view that shall be removed. In the `CREATE` operation, however, it is used to define the CPM that shall serve as basis for the new process view. In addition to the `resourceID`, the `CREATE` requires a `viewName` (line 7) that provides the name of the process view.

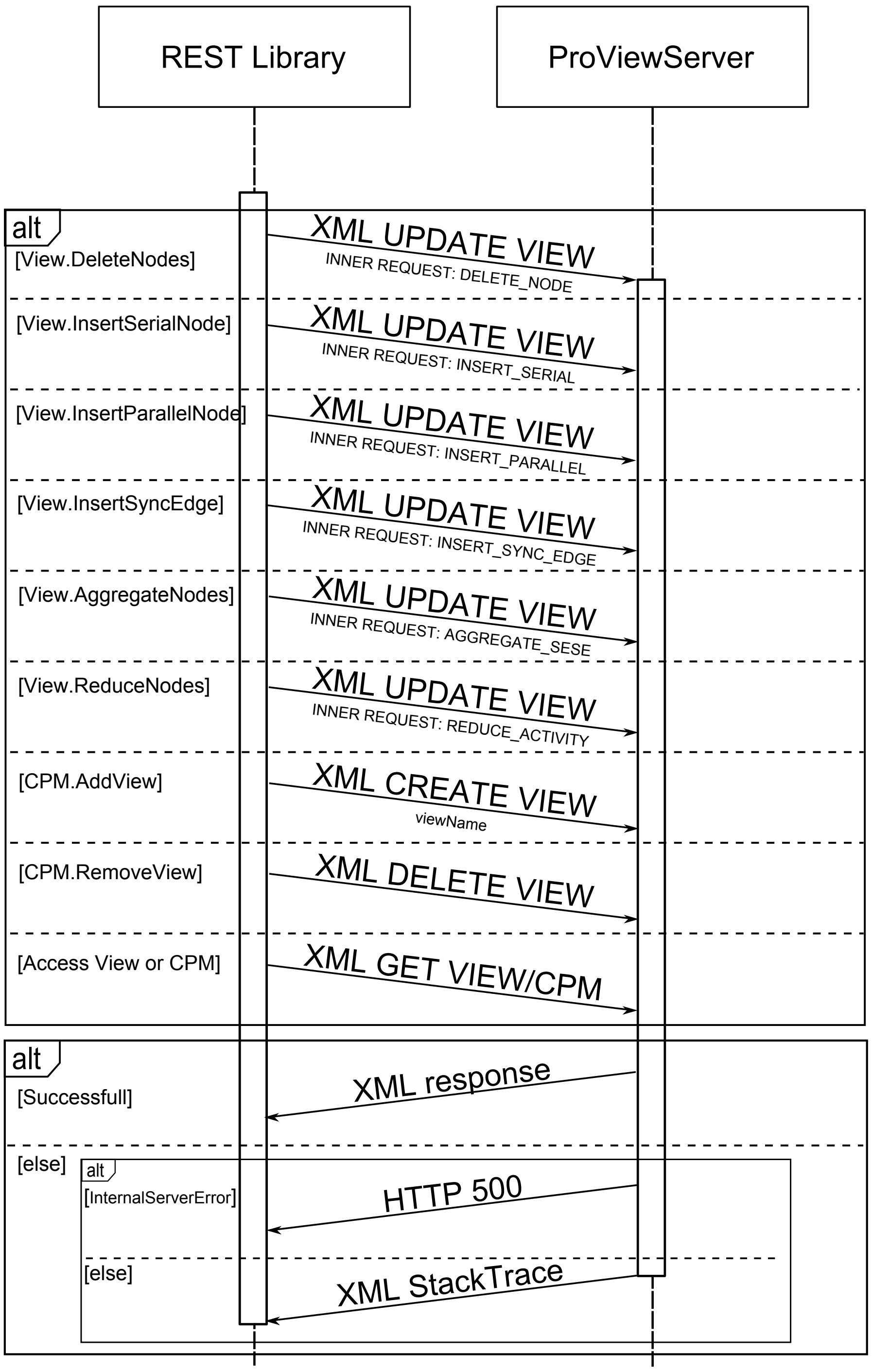


Figure 3.4: Synchronous Method Invocations.

UPDATE request operations are used to perform changes to the schema of a process view. Therefore, the `Inner Request` specifies the required update operation on the process view (line 7). Since the `Inner Request` is written in XML, the `payloadFormat` has to be set respectively (line 5). Furthermore, `resourceID` tells the proViewServer which process view to change.

Part 2: The `Inner Request` (cf. Listing 3.3), only used by `UPDATE` queries is part of the change request and gets XML-escaped before being inserted into the outer request.

```
1  <?xml version="1.0" ?>
2  <de.uniulm.proView.api.entity.view.operationset.(
      ViewUpdateOperation | ViewCreateOperation)>
3   <op class="de.uniulm.proView.api.entity.view.changeset.(
       UpdateChangeOperation|CreateChangeOperation)">
4      (INSERT_SERIAL|INSERT_PARALLEL|INSERT_SYNC_EDGE|DELETE_NODE)|
5      (REDUCE_ACTIVITY|AGGREGATE_SESE)
6   </op>
7  [<optionSet>
8     <entry>
9      <string>nodeName</string>
10     <string>name</string>
11    </entry>
12  </optionSet>]
13  <nodeSet>
14    <int>id_1</int>
      ...
15    <int>id_n</int>
16  </nodeSet>
17 </de.uniulm.proView.api.entity.view.operationset.(
      ViewUpdateOperation | ViewCreateOperation)>
```

Listing 3.3: Inner Request XML Format

Listing 3.3 shows the `Inner Request` and is categorized into two different types of operations, `ViewCreateOperation` and `ViewUpdateOperation`, which are described below. Both types use `nodeSet`[1] to specify the activities affected by the change (line 13).

[1]set of `Integer` IDs that represent a set of activities

`ViewCreateOperation` offers the support for `ViewCreateOperation` that are necessary to create a process view from a CPM (cf. Section 2.1.2) and is available by using the root element `de.uniulm.proView.api.entity.view.operationset.ViewCreateOperation` (line 2). The proViewServer expects the operation class `de.uniulm.proView.api.entity.view.changeset.CreateChangeOperation` for this request type (line 3). This operation enables all *REDUCE_SESE* and *AGGREGATE_SESE* operations whereas *REDUCE_SESE* performs a reduction and *AGGREGATE_SESE* an aggregation on the process view (cf. Section 2.1.2).

`ViewUpdateOperation` is used to make elementary update operations to a process view (and possibly changes to the underlying CPM). It can be accessed by using the root element `de.uniulm.proView.api.entity.view.operationset.ViewUpdateOperation` (line 2). The proViewServer expects the operation class `de.uniulm.proView.api.entity.view.changeset.UpdateChangeOperation` for this kind of update request (line 3). Several elementary view update operations are supported:

- *INSERT_SERIAL* inserts a serial activity between two activities.
- *INSERT_PARALLEL* inserts an activity in parallel to a set of activities represented by two activities. To achieve the parallel activity, the *INSERT_PARALLEL* command uses an *ANDsplit* and an *ANDjoin* gateway.
- *INSERT_SYNC_EDGE* inserts a synchronization edge between two activities on parallel branches.
- *DELETE_NODE* removes a set of activities.

3.2.4 The Reply to a Change View or CPM -Request

After the proViewServer processed the request, it replies with an XML file through a HTTP POST TCP tunnel that provides additional information depending on the type and the success of the response. The different reply types are induced by either successful or unsuccessful `CREATE`,`DELETE`, `UPDATE`, and `GET` requests.

Successful CREATE, DELETE and UPDATE requests If a `CREATE,DELETE, and UPDATE` request succeeds, the proViewServer returns a simple XML file (cf. Listing 3.4) repeating the request (line 2), and stating that the operation was successful (line 9). Additionally, it provides the computation time. The `result` tag is not significant.

```
 1 <de.uniulm.proView.api.entity.rest.Response>
 2   <request>
 3     <requestOperation>CREATE|DELETE|UPDATE</requestOperation>
 4     <resourceID>ID</resourceID>
 5     <payloadFormat>XML</payloadFormat>
 6     <resourceType>CPM|VIEW</resourceType>
 7     <query>...</query>
 8   </request>
 9   <status>SUCCESSFUL</status>
10   <duration>0.0</duration>
11   <result>
        ...
12   </result>
13 </de.uniulm.proView.api.entity.rest.Response>
```

Listing 3.4: Successfull Response to CREATE,DELETE, and UPDATE operation.

Successful GET Request: Since the `GET` reply contains more information than the other replies, the XML file structure for this response is different. It is designed to specify a process model as described in Section 2.1.

```
<?xml version="1.0"?>
<template id="ID" version="16" xmlns:xsi="..." xsi:
   schemaLocation="..." xmlns="...">
<name>CreditApplication</name>
 ...
<nodes>
 <node id="n1">
  <name>node name</name><description /><staffAssignmentRule/>
  <autoStart>false</autoStart>
 </node>
 ...
</nodes>

<dataElements>
 <dataElement id="d0">
   <type>(STRING|INTEGER|USERDEFINED)</type>
```

```
14      <name>element name</name><description/>
15      <identifierID>string</identifierID>
16      <isPublic>false</isPublic>
17    </dataElement>
      ...
18  </dataElements>
19
20  <edges>
21    <edge sourceNodeID="n_b" destinationNodeID="n_a"
22           edgeType="(ET_CONTROL|ET_SYNC|ET_LOOP)">
23      <edgeType>(ET_CONTROL|ET_SYNC|ET_LOOP)</edgeType>
24    </edge>
      ...
25  </edges>
26
27  <dataEdges>
28    <dataEdge connectorID="0" dataElementID="n_n" nodeID="d_m"
29              dataEdgeType="(WRITE|READ)">
30      <dataEdgeType>(WRITE|READ)</dataEdgeType>
31      <isOptional>(true|false)</isOptional>
32    </dataEdge>
      ...
33  </dataEdges>
34
35  <startNode>n_s</startNode>
36  <endNode>n_e</endNode>
37
38  <structuralData>
39    <structuralNodeData nodeID="n_m">
40      <type>(NT_NORMAL|NT_XOR_SPLIT|NT_XOR_JOIN|NT_AND_SPLIT
41             |NT_AND_JOIN|NT_STARTFLOW|NT_ENDFLOW|NT_XOR_SPLIT
42             |NT_ENDFLOW|NT_STARTLOOP|NT_ENDLOOP)</type>
43      <topologicalID>top id</topologicalID>
44      <branchID>branch id</branchID>
45      <correspondingBlockNodeID>n_i</correspondingBlockNodeID>
46    </structuralNodeData>
       ...
47  </structuralData>
48  </template>
```

Listing 3.5: Successful Respond to GET Request

Listing 3.5 shows an example of an successful GET Request response. The response provides a list of `nodes` (lines 4-9), `edges` (line 20), `dataElements` (lines 11-18), `dataEdges` (lines 27-33), and `structuralData` (lines 38-47). The list of `nodes` describes how many nodes are in the process model and how they are labeled. The edges `ET_CONTROL` and `ET_LOOP` are used to connect individual nodes of the process model to build an unidirectional graph. `ET_SYNC` edges are used between nodes of parallel branches to provide synchronization. `StructuralData` elements are used to further particularize nodes and provide `type` (line 40) information that specifies the exact node type. Unlike the process model definition in Section 2.1, the list of `nodes` and `edges` do not contain any `data flow` information. Instead, it is stored in the lists of `dataElements` and `dataEdges`.

Unsuccessful Request: When a request fails, the proViewServer either returns a HTTP 500 internal server error or an XML file containing the reason for the error (cf. Listing 3.6). The latter is parsed by the «REST Library» and a `RestException` object is created. Either way, the observers of the «REST Library» are notified of the exception and get either the unmodified `WebException` or the `RestException`.

```
<de.uniulm.proView.api.entity.rest.Response>
  <request>
    <requestOperation>(CREATE|DELETE|UPDATE)</requestOperation>
    <resourceID>ID</resourceID>
    <payloadFormat>XML</payloadFormat>
    <resourceType>(CPM|VIEW)</resourceType>
    <query>
       ...
    </query>
  </request>
  <status>FAILED</status>
  <duration>0.0</duration>
  <e class="java proViewServer exception class">
    <stackTrace>
       <trace>java stracktrace line</trace>
        ...
    </stackTrace>
  </e>
</de.uniulm.proView.api.entity.rest.Response>
```

Listing 3.6: Unsuccessful Request.

The structure of cf. Listing 3.6 has similarities to the one of a successful request (cf. Section 3.2.4, Listing 3.4). However the `status` is different (line 10) as well as an additional tag is included (line 12). The latter encapsules a full Java stack trace.

3.3 Receiving Process Model Changes

Although the used REST communication protocol supports requesting process models (cf. Section 3.2.3), it does not support server side push notifications. This is essential when an external change on one of the process models occurs. Therefore, the «REST Library» has to rely on methods like polling or user initiated manual updates. These methods acquire the list of process models and compare the version numbers with the stored ones (cf. Section 3.2.2). The developers of the proViewServer might consider to add support for a notification via `HTTP 1.1 Transfer-Encoding:chunked`. With this, it would be possible to send short update notifications to the connected clients so that they can update their process models. The *ProViewKinect* (cf. Section 6) uses the «REST Library» with a five second update interval to keep its process models up to date.

3.4 Parse Graph Changes

When a change was requested (cf. Section 3.2.3) or received (cf. Section 3.3), the «REST Library» obtains the new process model from the proViewServer. The idea behind parsing this process model is to inform the observers of the «REST Library» not only about that a change occurred but also what has changed. Hence, the observers need only to update the modified parts of their process model representations.

As described, the changed process model is received by a successful GET request (cf. Section 3.2.4).

The «REST Library» uses three lists for this purpose. The first list contains the deleted nodes. The second is used to track the new nodes. The last contains the updated nodes at the end of the computation.

To fill the lists, the algorithm parses the XML file for the nodes (Listing 3.7). If a node exists in the list of old nodes, it is removed (line 12). If it was changed, it is copied into the list of updated nodes (line 19). If not, the algorithm creates a new node and stores it in the list for new nodes (line 23).

At the end, the first list contains all elements that existed in the old but not in the new graph, the deleted nodes. The second list contains all new nodes since all new nodes are added to it on creation. The third list contains all updated nodes since the nodes in this list were found in the list of old nodes but with different values. These three lists are then sent to the observers so that they can update their process models.

A modified version of this algorithm is used to track changes on the edges.

```
1  // contains all old nodes, at the end only the deleted nodes
2  Dictionary<int, Node> oldNodes = model.NodeDictionary;
3  // empty, will contain all new nodes
4  List<Node> addedNodes = new List<Node>();
5  // empty, will contain all reused nodes
6  List<Node> updatedNodes = new List<Node>();
7
8  foreach (XmlNode n in nodes){
9      if (oldNodes.ContainsKey(id)){
10         // reuse old node
11         node = oldNodes[id];
12         oldNodes.Remove(id);
13
14         // check if the node has changed
15         if (!node.Name.Equals(name) || node.Type != type ||
              node.TopologicalId != topologicalId || ...){
16             // update values
17             node.UpdateValues(name, type, topologicalId, ...);
18             // add node to the updated node list
19             updatedNodes.Add(node);
20         }
21     }else{
22         // create new node
23         node = new Node(id, name, type, topologicalId, ...);
24         addedNodes.Add(node);
25     }
26 }
```

Listing 3.7: Parse node changes

3.5 Using the REST Library

Listing 3.8 shows an example on how the «REST Library» may be used.

```
class Program: IObserver<Event>
{
 public Program(){
  Rest r = Rest.Instance;

  r.Subscribe(this);
  r.SetUpdateTimer(5000);

  foreach (Cpm cpm in r.CpmList)
  {
    Node[] n=cpm.Nodes;
    foreach (var view in cpm.Views)
    {
      n = view.Nodes;
      if(n.Count()>3)
        view.DeleteNodesAsync(new[] {n[2], n[3]});
    }
  }

  if (r.CpmList.Count() > 0)
    r.CpmList[0].AddView("my new view");
 }

 public void OnNext(Event value)
 {
   Console.WriteLine("Event " + value);
 }
 public void OnError(Exception error)
 {
   Console.WriteLine("Exception " + error);
 }

 public void OnCompleted()
 {
   throw new NotImplementedException();
 }
}
```

Listing 3.8: Example usage of the «REST Library».

To use the library, the user needs to subscribe to it (line 6). Therefore, the subscriber needs to implement the `IObserver<Event>` interface. The interface provides the methods `OnNext(Event)`, `onError(Exception)`, and `OnCompleted()`, which are used by the «REST Library» to notify the subscriber about errors and changes on the process models. The `OnCompleted()` is not used and may be ignored although it has to be implemented as part of the C# observer pattern.

The `SetUpdateTimer` call (line 7) sets the «REST Library» to poll the server. This is necessary to receive any external modifications to the process models stored in the proViewServer (cf. Section 3.3).

Lines 9-18 iterate through all CPMs and their process views and tries to delete the second and third node from the process views. If the operation succeeds, the `OnNext` method (line 24) gets called with a `GraphChangedEvent` holding the information on how the graph has been changed. If not, then the `OnError` method (line 28) gets called containing the description why the operation has failed. There are two types of errors. A failed request where the proViewServer returns an XML file explaining the error or an unforeseen error (e.g., 500 internal server error). When the first case applies, `OnError` is called with a `RestException`. Otherwise, if an unforeseen error occurred, it is called with an `ExceptionCapsule` exception. The `ExceptionCapsule` exception encapsules the exception of the error and contains additional information.

Note that the invocation of `DeleteNodesAsync` can be replaced by `DeleteNodes`, if a synchronous (blocking) call is required. If the synchronous variant is used, it is important not to execute it from the GUI thread or the application may freeze for 10 seconds.

The «REST Library» supports the deletion and creation of process views. As example, line 21 adds a new process view to the first CPM in the CPM list. If the operation succeeds, `OnNext` is called with a `ModelChangeEvent` and `EventType` set to `Created`. On fail, the `OnError` method is called as described above.

4

Processing Process Models

After a process model is obtained by the «REST Library», it has to be processed before it can be displayed or interacted with. Therefore a displayable process model computed from a list of nodes and edges is required. For that reason, an algorithm is required that calculates the position and sizes of the process model elements (i.e., nodes and edges). This algorithm is divided into two parts. The first part, described in Section 4.1, separates the graph into *boxes*. The second part, described in Section 4.2, calculates the actual positions for painting the process model. Section 4.3 deals with mapping the user input onto the process model.

4.1 Creation of Process Model Blocks

The *CalculateBlocks* algorithm is the first step to achieve a displayable and intractable process models.

The basic idea of the algorithm, used in the «Kinect Process Modeling» component (cf. Section 6.1), is to divide the graph into several cascading boxes called process *blocks*. Each time a path in the process splits into multiple other paths, a new block for each path is created which is closed when its path joins with another path. The result is shown in Figure 4.1.

The algorithm is a based on [Bür12], which also uses blocks to layout, but supports two different types of blocks as well as multiple methods to evaluate the different block types.

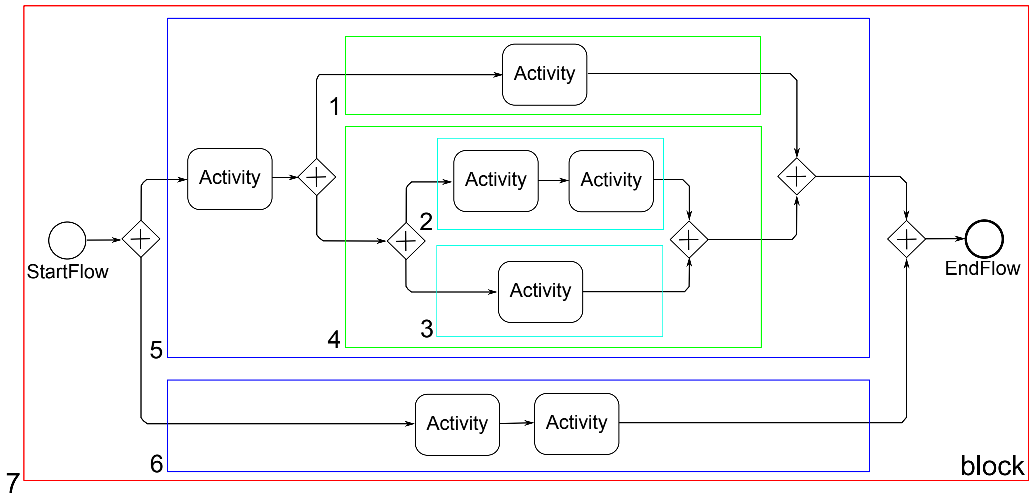


Figure 4.1: Result of Graph Parsing Algorithm

The algorithm that creates the cascaded process blocks is recursive. Figure 4.1 demonstrates the order (1-7) in which the blocks are calculated. Note that all inner blocks have to be calculated before their respective outer block. The blocks (2) and (3), for example, needs to be calculated before (4),(5), and (7) because their width and height is decisive for them.

Additionally to the blocks, the algorithm calculates the horizontal offset of every node in the process model relative to its parent block.

```
private Block CalculateBlocks(Node currentNode,Block parent, int hBlockOffset){
  int hUnitOffset = 0; //offset relative to the block horizontal offset
  Block block = new Block(hBlockOffset, parent);
  List<Block> allInnerBlocks = new List<Block>();

  while (currentNode.Type != NodeTypes.join){
    // create graph element and add it to the current block
    GraphElement graphElement = NodeToGraphElement(currentNode, hUnitOffset);
    block.addGraphElement(graphElement);
    hUnitOffset += graphElement.UnitsWidth + Block.hUnitSpace;

    List<Edge> edges = currentNode.ControlEdges;

    if (edges.Count() == 0) break;
    if (edges.Count() == 1){
        currentNode = edges[0].To;
    }else{
        // recursively parse inner blocks
        List<Block> innerBlocks = (from edge in edges select CalculateBlocks(edge.To, hUnitOffset + hBlockOffset, block)).ToList();

        // add innerBlocks to the list of all inner blocks
        allInnerBlocks = allInnerBlocks.Concat(innerBlocks).ToList();

        // add inner blocks sizes to the block
        int totalUnitWidth = block.AddSizes(innerBlocks);
        // add max inner block width to the horizontal offset
        hUnitOffset += totalUnitWidth;

        List<Edge> lastEdges = innerBlocks[0].LastNode.ControlEdges;

        // get the node after the end of the inner block
        currentNode = lastEdges[0].To;
    }
  }
  block.InnerBlocks = allInnerBlocks;
  return block;
}
```

Listing 4.1: Caclulation of Process Model Blocks.

First the `CalculateBlocks` algorithm initializes its variables (cf. Listing 4.1). Variable `hUnitOffsets` is used to track the horizontal progress inside of a block in order to tell the individual nodes (line 8) and inner blocks (line 19) where they are located horizontally. This offset begins at zero (line 2), because there are no nodes or inner blocks at the beginning. Then, the block object is created with the `hBlockOffset` parameter (line 3). This parameter, used to set the absolute horizontal offset of the block, is later required for the position calculation (cf. Section 4.2). Next, `currentNode` parameter (line 1) that has to be set to the block's start node, i.e., the first node of the block.

After initialization, the calculation starts in line 6 with a while loop, running until reaching the next join node since join nodes mark the end of a block. Inside of the loop, a `GraphElement` is created for the current node and added to the actual block (line 9). Such a `GraphElement` encapsules the node as well as additional variables like horizontal offset, width, height, and location of the node. Then, `hUnitOffset` is increased, because the next node needs to be positioned further to the right (line 10).

Further, the algorithm proceeds to the next process node. Three cases might occur: First, the node has no outgoing edges. Then, the block is terminated (line 14). Second, exactly one outgoing edge exists and, therefore, one subsequent node. The node is then taken as the new `currentNode` and is added to the block in the next loop iteration. Third, `currentNode` has multiple outgoing edges. Then, a new block is created for each path, i.e., the method is recursively called (line 19). The parameter of the call includes the edge of one path, the new `hBlockOffset` calculated from the old, the current `hUnitOffset` and the current block which serves as parent for the nested blocks. After the calculation of the inner blocks, they are gathered to a list (line 22). This list is held by the actual block to track nested blocks. Then, the algorithm adds the sum of the inner block heights and widths to the block by invoking `AddSizes` method (line 25). This ensures that the block has enough space to hold the inner blocks. The `AddSizes` method additionally returns the maximum of the inner block widths, which is required to increase the local `hUnitOffset` (i.e., the following nodes are not placed inside of one of the inner blocks, but after).

Last, the new `currentNode` is calculated, which represents the node that follows directly after a join node (i.e., the node after a block is the node after the last node of the block (line 32)).

Runtime Analysis: It is important that the algorithm (cf. Listing 4.1) is efficient to handle huge process models in an acceptable amount of time. Therefore we take a look at the complexity of the algorithm.

Definition 3 (Nodes and Blocks)**.** *Let N be the nodes and $B \subseteq N$ a block and S the set of all blocks then rule $\forall n \in N \exists! B \in S$: n ∈ B applies.*
Note that Definition 3 ignores that it is possible that for $B_1 \in S$, $B_2 \in S$,$|B_1| < |B_2|$ and $B_1 \in B_2$ thus $n \in N, n \in B_1 \Rightarrow n \in B_2$ (i.e., a node in cascading blocks is part of all parent blocks) because it has no effect on the computation compexity of the `CalculateBlocks` *algorithm.*

Definition 4 (Parallel Paths)**.** *Let n_i, $n'_j \in N$ and p, p' be two paths with $p = n_1, n_2, \ldots, n_{y-1}, n_y$ and $p' = n'_1, n'_2, \ldots, n'_{z-1}, n'_z$. Then p is parallel to p' when n_1 and n'_1 are the same split gateways, n_y and n'_z the same join gateways and the therm $\forall u \in \{2, \ldots, y-1\}$ $\forall v \in \{2, \ldots, z-1\} : n_u \neq n'_v$ applies. Let PP then be the set of all parallel paths in a process model.*

The algorithm consists of just one loop. However it contains recursive calls that increases the complexity. To measure the complexity we, therefore, look at the nodes accessed since every loop iteration accesses exactly one node. Since every node is in one block (cf. Definition 3), the wrong conclusion, that computation complexity is $\mathcal{O}(|N|)$, can be made. However, the loops inspect the join nodes at the end of a block for each path. The amount of additional inspections is equal to the amount of parallel paths (cf. Definition 4) in a block. The complexity of the algorithm is therefore the amount of loop iterations added to the amount of additional inspections, $\mathcal{O}(|N| + |PP|)$.

4.2 Calculation of Node Positions

CalculatePositions (cf. Listing 4.2) is the second step to achieve a displayable and intractable process model.

The algorithm calculates x- and y-positions of the nodes. Therefore, it iterates through every block and assigns node positions based on the `unitsToPixel` variable that is used to scale the resulting graph of a process model.

```
private void CalculatePositions(Block block, double
   unitsToPixel)
{
 // set bounds and position of block
 block.X = (int)(block.AbsoluteHUnitOffse * unitsToPixel);
 block.Y = (int)(block.AbsoluteVUnitOffset * unitsToPixel);
 block.Width = (int)(block.UnitWidth * unitsToPixel);
 block.Height = (int)(block.UnitHeight * unitsToPixel);

 int yUnitMiddle = block.YUnitStart;
 int xUnitStart = block.AbsoluteHUnitOffset;

 foreach (GraphElement graphElement in block.Elements)
 {
    // sets position of the graph element
    graphElement.SetPos((int)((xUnitStart + graphElement.
       HUnitOffset) * unitsToPixel),
                        (int)((yUnitMiddle - graphElement.
                           UnitsHeight / 2) * unitsToPixel));

    // calculate size
    graphElement.SetSize((int)(graphElement.UnitsWidth *
       unitsToPixel), (int)(graphElement.UnitsHeight *
       unitsToPixel));
 }

 // recursively calculate positions of inner blocks
 foreach (Block b in block.InnerBlocks)
 {
     CalculatePositions(b, unitsToPixel);
 }
}
```

Listing 4.2: Calculation of Node Positions in a Process Model.

The Algorithm has two input parameters (line 1). First, parameter "`block`" tells the algorithm which block to work with and is set to the inner blocks when called recursively. Second, parameter "`unitsToPixel`" which is used to scale the graph.

The algorithm starts with the calculation of constants representing the actual pixel position and size of the blocks (lines 4-7). These constants are later required to map user inputs onto the nodes (cf. Section 4.3).

Next, the vertical center (line 9) and horizontal start point (line 10) of the block are calculated. The following node positions are based on these values ensuring that the nodes are positioned vertically centered in their respective blocks.

The loop, starting in line 12, iterates through all nodes of the block and sets their positions (line 16) as well as their sizes (line 19). The position calculated is based on the start position of a block and the nodes offset. The nodes sizes are based on the unit size of the nodes converted to pixels.

Finally, the algorithm recursively calls itself to calculate the positions of all inner blocks (lines 23-26).

Runtime Analysis: To handle huge process models, the runtime complexity of the algorithm is essential. The algorithm CalculatePositions has similarities to the one presented in Section 4.1. The main difference is that it does not have the additional requirement to take care of the join nodes and, thus, the runtime is $\mathcal{O}(|N|)$.

4.3 Collision Detection

The algorithms in Section 4.1 and 4.2 enables us to paint the process models on the GUI but they do not support handling user interaction. Therefore another algorithm is required that maps user input onto the displayed process model.

The `CollisionDetection` algorithm uses the x- and y-coordinate of the user input (e.g., mouse click, gesture input) and calculates the node on which the input is performed. The naive way to get the node would be to cycle through all nodes of the process model and check if the coordinate is within a node. However, worst-case complexity with this method is $\mathcal{O}(|\mathrm{N}|)$, where $|N|$ is the set of all nodes (cf. Definition 3). This is not acceptable for large business models.

```
1  private GraphElement CollisionDetection(int x, int y, Block
      block){
2   foreach (Block innerBlock in block.InnerBlocks)
3     if (innerBlock.ContainsPoint(x, y, UnitsToPixel))
4     {
5       return CollisionDetection(x, y, innerBlock);
6     }
7   foreach (GraphElement graphElement in block.Elements)
8     if (graphElement.ContainsPoint(x, y))
9     {
10      return graphElement;
11    }
12  return null;
13 }
```

Listing 4.3: The C# algorithm checks if the x and y parameter are on a graph element.

Algorithm `CollisionDetection` presented in Listing 4.3 iterates through all blocks and recursively looks if it contains the searched node. If the node was not found in one of the inner blocks, it is searched in the block itself.

The algorithm starts with the root block, i.e., the block that has no parent block. It first tests the inner blocks whether they contain the respective coordinate (lines 2-6). If yes, the algorithm recursively calls itself with the block containing this coordinate. Otherwise, it checks the nodes of the actual block (lines 7-11) and returns the node that contains the coordinate. If no node is found, it is assumed that there is no node that matches the coordinate (line 12).

Generally the algorithm uses the advantage that nodes and blocks are aligned as a tree, in which nodes are leafs and blocks are inner nodes. The algorithm uses a Depth-first search to find a path from the root to one leaf. If no fitting node exists, the search gets stuck at one inner node where it can not find a fitting leaf.

Worst-case complexity of the algorithm is $\mathcal{O}(|B|)$ where B is the set of all blocks. Since $|B|$ is usually smaller than $|N|$ in process models, the recursive algorithm is to be preferred over the naive algorithm.

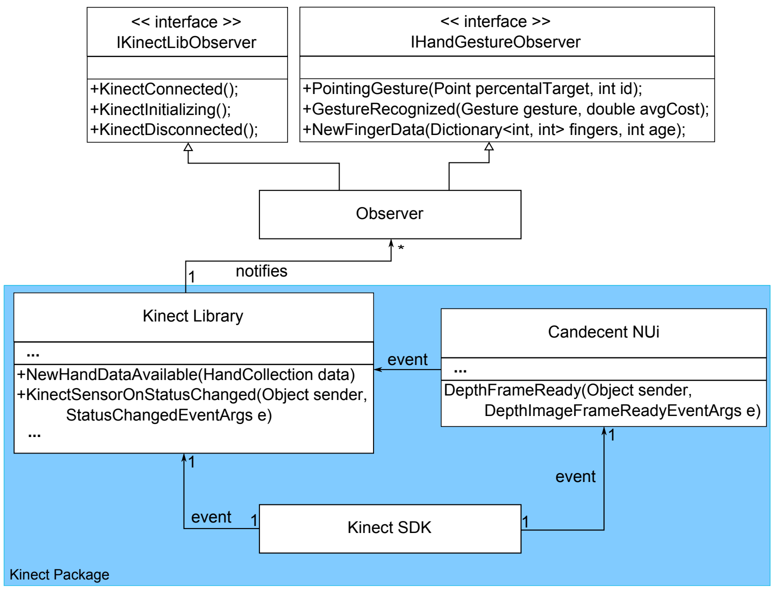
5 Kinect Library

The «Kinect Library» is the third component of the architecture. It abstracts the communication with the *Kinect SDK* and offers a user interface to enable gesture-based user input. To achieve this, it relies on the *Candescent NUI* (cf. Section 5.3) library that offers hand and finger recognition [Ste13] and uses algorithms like DTW (cf. Section 2.3) to parse gestures from finger data.

Section 5.1 explains the structure of the «Kinect Library». Section 5.3 explains a library that is used to detect the users hand and its fingers. Section 5.4 describes how the detected fingers are processed to reduce fuzziness of the input. Section 5.4.2 describes how the finger data is processed in order to use the users fingers as a pointing device. Section 5.5 describes how finger data is processed in order to calculate gestures for process model manipulation.

5.1 Structure

The *Kinect Package* consists of three components (cf. Figure 5.1). First, Kinect SDK [Win13b], providing a depth data stream and status information of the Kinect hardware. Second, Candescent NUI library [Ste13], offering hand and finger position information. Third, the observable «Kinect Library», providing gesture recognition and a three dimensional finger-based input system.



<< interface >>
IKinectLibObserver
+KinectConnected();
+KinectInitializing();
+KinectDisconnected();
<< interface >>
IHandGestureObserver
+PointingGesture(Point percentalTarget, int id);
+GestureRecognized(Gesture gesture, double avgCost);
+NewFingerData(Dictionary<int, int> fingers, int age);
Observer
*
1
notifies
Kinect Library
...
+NewHandDataAvailable(HandCollection data)
+KinectSensorOnStatusChanged(Object sender, StatusChangedEventArgs e)
...
event
Candecent NUi
...
DepthFrameReady(Object sender, DepthImageFrameReadyEventArgs e)
1
event
1
event
1
Kinect SDK
1
Kinect Package

Figure 5.1: «Kinect Library» Components

Kinect SDK is used to get the Kinect sensor's status information. On change, the Kinect SDK sends an event to the «Kinect Library» which notifies its observers by calling the corresponding method (i.e., KinectConnected, KinectInitializing or Kinect Disconnected). Further, Kinect SDK calls the Candescent NUI's `DepthFrameReady` method and informs it about new depth information from sensors of the Kinect. The Candescent NUI library then calculates the hand information (i.e., hand and finger position and vectors) and supplies it to the «Kinect Library». The «Kinect Library» calculates the gestures and finger information from the consecutive hand information and informs the observers of the «Kinect Library» about the recognized gestures, the amount of fingers on each hand, and where the user is pointing with his finger.

5.2 Microsoft Kinect

The Kinect was developed as a toy by Microsoft and was released in 2010. The Kinect has a depth sensor consisting of an infrared projector and an active infra red sensor obtaining depth information regardless of the ambient lighting. Additionally, it has a RGB camera and an array of microphones. The combination of the RGB and the depth camera is called RGB-D camera. The resolution of it is limited to 640x480 with a *field of view* (FOV) of 43° vertical and 57° horizontal. Technically a resolution of 1280x960 with the same FOV is possible but at the cost of frame rate [Win13a]. There are two versions of the Kinect. First, *Kinect for Xbox*, mainly used for gaming. Second, *Kinect for Windows*, used for development and supporting *near mode*. Near mode is a setting for the depth sensors to focus on tracking closer objects enabling a detection between 0.4 and 0.8m (cf. Figure 5.3) [Cra13]. The area covered in this range is about 90x85x50cm small.

In our use case the Kinect is mounted upside down at a height of about 1 meter so that its cameras point downwards (cf. Figure 5.2). This was done so that the user can brace his elbows on the working surface.

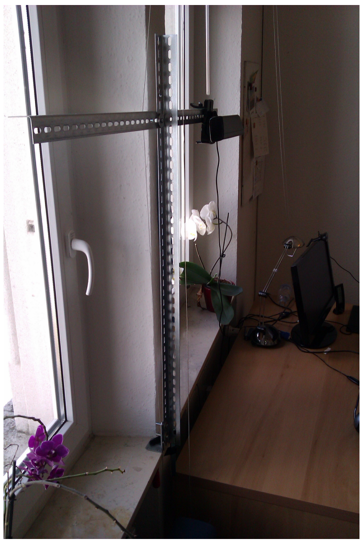


Figure 5.2: Kinect Mounted Upside Down.

5.3 Candescent NUI

The *Candescent NUI* (CCT NUI) library is used to calculate the 3D hand information from a raw Kinect depth footage obtained from the *Microsoft Kinect* SDK [Ste13, Win13b]. Since hand detection requires a high resolution [MPC12] and the library uses the limited resolution of 640x480 pixel, fingers can only be tracked at a range up to 1m. Therefore, Kinect hardware must support the near mode (c.f. Section 5.2, Figure 5.3).

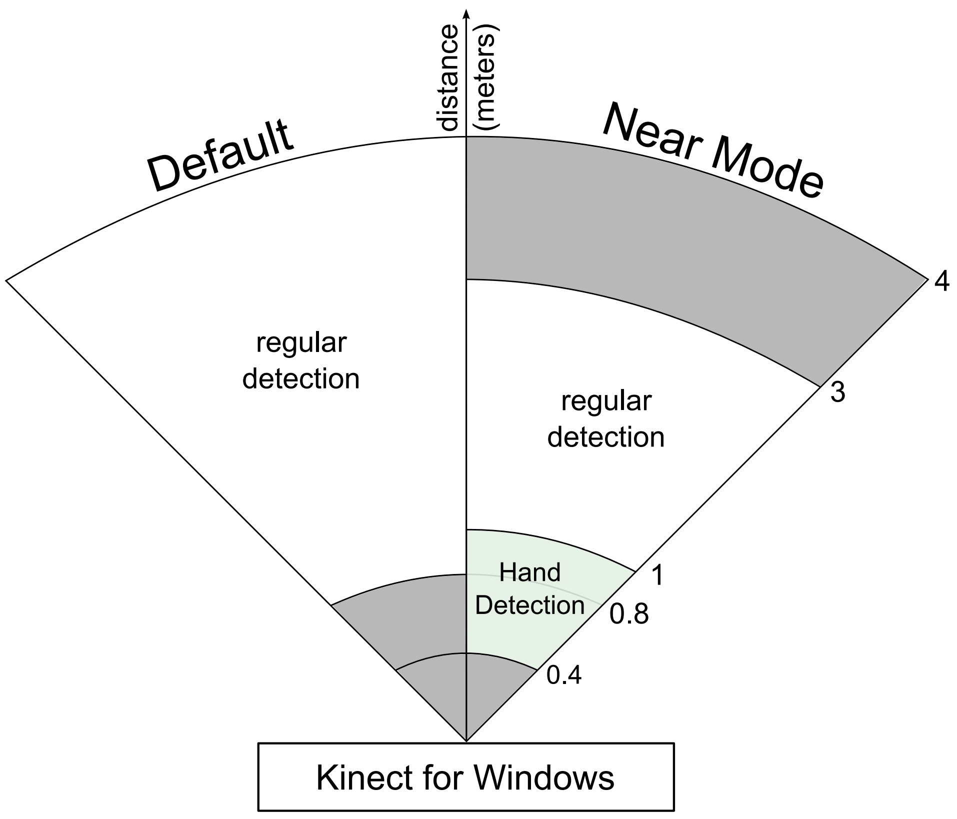


Figure 5.3: Default vs. Near Mode Hand Detection

To calculate the hand information the CCT NUI library applies a clustering algorithm generating clusters in the depth image. A limited depth range is used preventing cluster detections outside of the working area. After the clustering is done, CCT NUI searches hand and finger patterns in the cluster matrix. Using this method, CCT NUI can not distinguish between a hand and any hand-like objects and, therefore, false positives are possible.

The main advantage of the CCT NUI to other Kinect hand tracking libraries is that the CCT NUI does not require a coping process (i.e., the user does not have to hold his fingers in a certain position until they are detected) so that it can be used on-the-fly. Further more it does not require any additional hardware (e.g., a CUDA ready NVIDIA graphic card for GPU acceleration) other than the Kinect.

As disadvantage, the library is susceptible to noise and requires a lot of computation power (up to 3.4 GHz) running the clustering algorithm on every depth frame provided by the Kinect RGB-D camera.

5.4 Hand Tracking

This section addresses how the hand data offered by the CCT NUI library is processed in order to obtain viable finger count, position, and direction information.

5.4.1 Finger Count

The CCT NUI library offers finger count information (i.e., how many fingers are visible at a time). However, this information may be very inaccurate depending on several factors like the hands angle, sunlight nuisance and speed of the users hand. Even a slight movement of the tracked hand may result in different amount of fingers detected.

Three steps have been undertaken to improve this detection issue.

Step 1: Fix detection based on the coordinate: Sometimes there are wrong detections in the area below 15 pixels on the y-axis. In Figure 5.4, four fingers (F1-F4) are correctly detected, the fifth finger (F5) is a bad detection. These false detections occur when the user's arm enters the picture at a certain angle and is partly recognized as a finger. However, it is possible that a recognition in this area is valid, i.e., the user has his hands in the area between 0 and 50 pixels and there is a finger located below 15 pixels. To solve this issue, the CCT NUI library has been altered in order to ignore all detections below 15 pixels, iff, another finger is detected at 50 pixels and above. Obviously, it is easier to ignore all detections below 15 pixels but this creates a detection "dead zone" for that area.

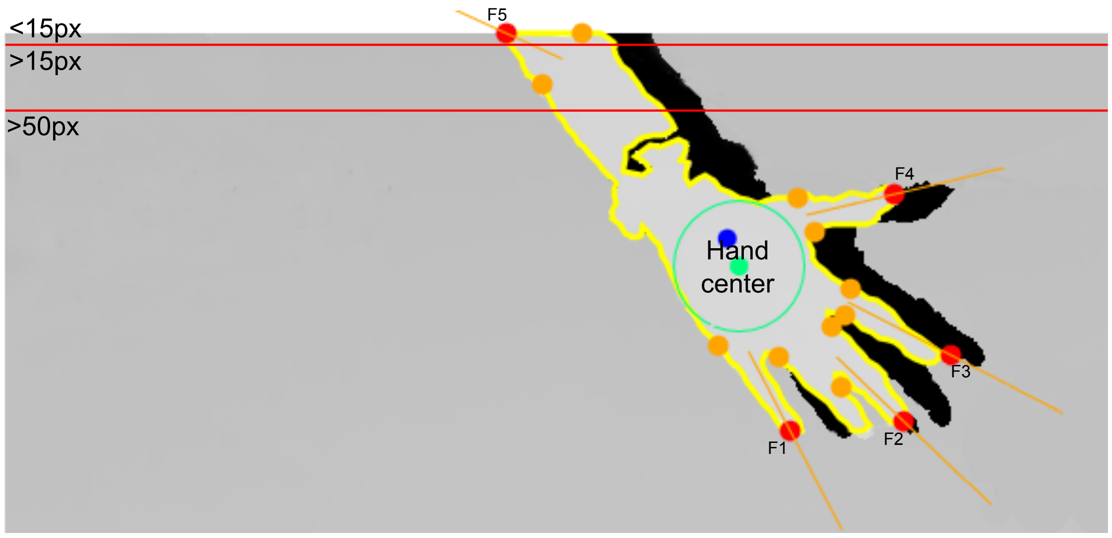


Figure 5.4: Bad Finger Detection

Step 2: Fix non-logical Finger Counts: The CCT NUI library may report finger counts greater than five. When this happens, the «Kinect Library» invokes the method "`CorrectData`" that has been added to the CCT NUI library. This greedy method calculates all possible distances between finger points of the last correct detection and current non-logical detection. Then, the fingers are ordered by these distances and the first five fingers of the non-logical detection whose distance is minimal are selected. Put in another way, the method removes the additional fingers by predicting the ones that are most likely incorrect.

Step 3: Bad Values: In some cases the CCT NUI library suddenly reports wrong finger counts. This errors are difficult to pinpoint, because they are dependent on several factors (cf. Section 5.4.1). The most common error is CCT NUI library suddenly returns that no fingers are found even though the user is still showing his fingers. To decrease this problem, the «Kinect Library» uses a data structure that is similar to a branch prediction table (cf. Listing 5.1). It counts how often a finger count was detected in a row and in case of a change, calculates how likely the new value is right or wrong. The closer the new value is to the current value, the more likely it is no error (line 12). A zero finger detection is a special case, because it is more likely an error and, therefore, the algorithm decreases the likelihood of other fingers only by a low percentage (line 9). If a value appears again, the likelihood of the reoccurrence increases. The algorithm illustrates this by doubling the occurrence likelihood factor up to the `MaxOccurenceCount` boundary (line 5).

```
1  for (int finger = 0; finger < _fingerOccurence[id].Count();
      finger++)
2  {
3   if (finger == data.FingerCount){
4     // increase finger occurence likelihood
5     fingerOccurence[id][finger] = Math.Max(1,Math.Min(
         MaxOccurenceCount, fingerOccurence[id][finger] * 2));
6   }else{
7     if (data.FingerCount == 0){
8       // zero detections are often mistakes
9       fingerOccurence[id][finger] *= 0.95;
10    }else{
11      // reduce the finger occurence likelihood
12      fingerOccurence[id][finger] *= (1 - 0.1 * Math.Abs(finger
           - data.FingerCount));
13    }
14   }
15 }
```

Listing 5.1: change finger possibility

After the algorithm finished calculating the estimated finger count, it calls method "`CorrectData`" in the CCT NUI library that was explained in the last paragraph. This method also handles the case where less fingers are found. In this case, it orders the fingers again, but selects the ones that are further away, i.e., the fingers that are most likely missing.

5.4.2 Finger Pointing

Finger Pointing system enables users to point with one Finger on a distant projected surface and the system calculates the respective x- and y-position and, additionally, the percental distance to the targeted surface (cf. Figure 5.5). The decision to use finger tracking instead of the more accurate hand tracking was made because it reduces the fatigue since it reduces the distance the user has to cover for an input. The detection

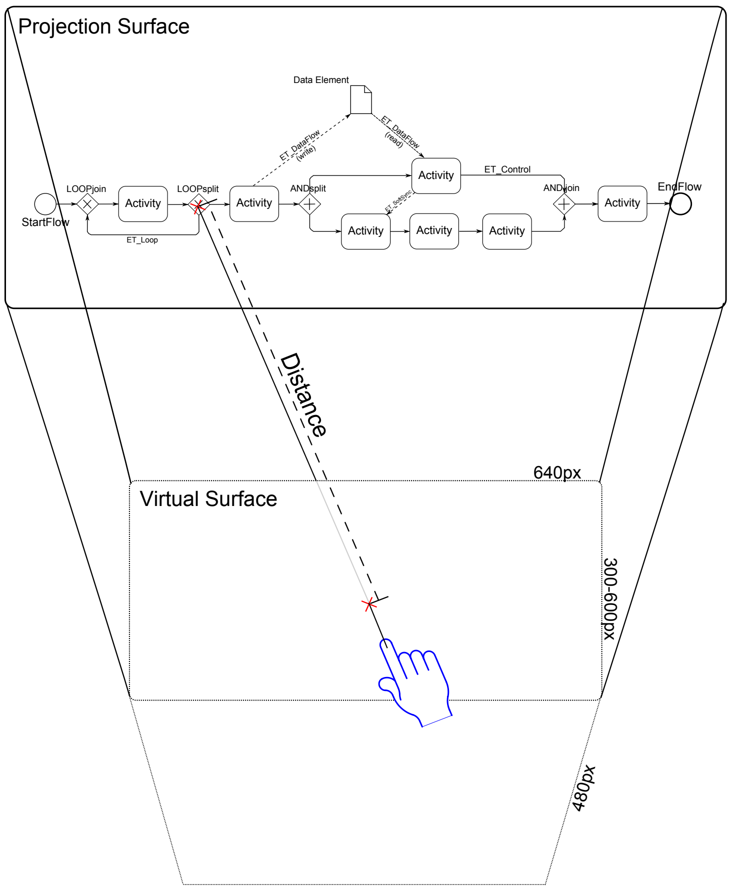


Figure 5.5: Finger Pointing

is written robust enough to work even if two fingers are detected instead of one. In this situation, it takes the finger that is closer to the distant surface.

The CCT NUI provides, therefore, the finger tip position and a vector representing the respective pointing direction. Calculating the percental x- and y-coordinate, the

«Kinect Library» takes the intersection of the vector (with the position as origin) with the projection surface and calculates the percental value from the x- and y-coordinate of the intersection. Receiving the distance, however, more effort is required. First, a virtual surface is defined that is located at the maximum z-value the finger may have (480 pixel). When a user is pointing to the projection image, the system additionally calculates the intersection with this virtual surface. It then takes the intersection point of the virtual and the real surface and calculates the euclidean distance between them. It then divides the value by the maximum distance to receive values between 0 and 1 and positions the cursor to that percental point.

Problems When used in practice, the calculated pointer seemed to jump randomly over the surface. The reason is that although a 100ms quantization is applied [LWZ$^+$09, Wik13] (i.e., 100ms of position values are arithmetically averaged) to the position values to reduce the noise, the percental x- and y-values vary about 20 to 50 percent. The Kinect SDK resolution seems to be not high enough for the CCT NUI library to determine an exact direction vector. To improve the accuracy, the quantization time has to be increased to at least one second with the drawback that this influences the user interaction experience in a negative way since input feels sluggish and, therefore, this type of input method has been abandoned.

5.4.3 Finger Position

Since the finger tip position data seems to be more accurate than the pointing vector, it is used to point on the surface (cf. Figure 5.6). The z-value is then calculated from the z-position of the finger relative to the maximal possible z-value of 480 pixels. The quantization is set to 600ms to eliminate the noise of the position data. This works well, but the sluggishness is still notable.

To decrease the delay, a *Scaled Radial Dead Zone* is introduced [Jos13]. This dead zone is applied to the percental difference between a new finger position and the previous position and does two things (cf. Figure 5.7): First, it eliminates all detections below 2%, since they are most likely noise. Second, it creates a gradient between 2% and 20%, that makes detections close to 2% weaker and detections close to 20% stronger. All detections above 20% are apparently no noise and can be taken into account directly. The previously mentioned delay could be reduced to 400ms with the dead zone. This delay is still noticeable, but is not as annoying as any delay greater than 600ms.

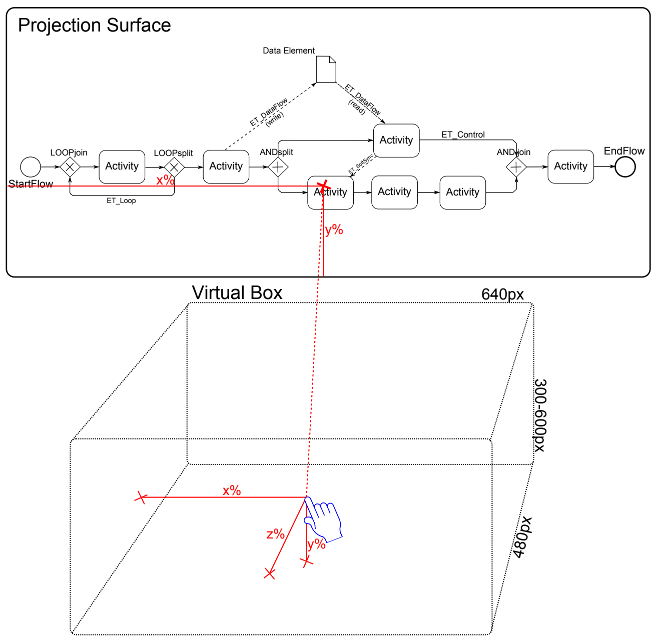


Figure 5.6: Finger Position

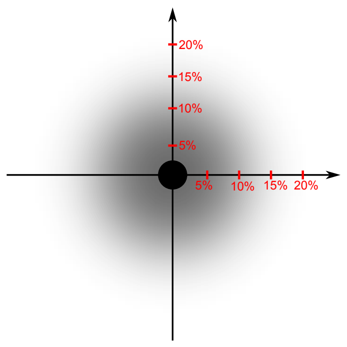


Figure 5.7: Scaled Radial Dead Zone

5.5 Gesture Recognition

The gesture[1] recognition can not work with fixed position values provided by the «REST Library» because a gesture consists of movements. The gesture recognition therefore requires a history that keeps track of the quantized discrete finger positions in order to calculate the x-,y- and z-movements of fingers. A gesture is then defined as a list of speed values over the time [PSH97, Rub91, LWZ^{+}09].

DTW is used to compare the speed values of the user input with predefined gestures (cf. Section 2.3). The algorithm needs two arrays for the computation. The first one

[1]We use "gestures" to refer to finger movements in the three dimensional space.

is predefined by the gesture. The second one is constantly created from the users finger movement speeds. Since we have multiple gestures with multiple data, we need multiple instances of the DTW algorithm. This may result in a high calculation cost. There are only few gestures in our case but there are cases with more gestures or where computation power is limited (e.g., on handheld devices or embedded systems). There, it might be wise to calculate only the DTW distance of gestures that make sense in the current application context.

When a gesture is recognized the system subtracts the time stamp where the last gesture was recognized from the current time stamp. If the resulting time difference is less than one second, it skips the gesture to prevent multiple recognitions at the same time. Otherwise, it informs the observers of the «Kinect Library» about the newly found gesture.

The problem with the used method of gesture recognition is that it is difficult to distinguish between a normal user interaction and gesture since the user can either select a node or perform a gesture with the same movement pattern. To solve this, a special condition is integrated which has to be fulfilled before gestures are recognized: A user has to show five fingers of his hand to the Kinect camera for about one second. This simple gesture enables the gesture recognition mode for a fixed time period.

5.5.1 Gesture Format Definition

Keeping gesture definition simple, the «Kinect Library» loads all XML files ending with "_gest.xml" out of a directory. Each XML file defines exactly one gesture, the action to be triggered, and additional meta data.

Listing 5.2 shows the definition of the nudge action gesture (i.e., the user moves his finger forth on the x-axis and then back). Though later replaced with a two finger click gesture, it is a good example on how gestures are defined. It starts with meta data (lines 3-6), which may be used to show the user the name of the recognized gesture for example. Followed by the `Action` triggered when the gesture is recognized (line 7). Right now the *click action*, to select, and all possible operations supported by proViewServer are allowed. The value of `Threshold` (line 8) is used to determine if the user input matched the gesture or not. If the average DTW distance calculated from the data sets by the DTW algorithm is less than the threshold then the gesture is recognized. Note that a low threshold might improve the recognition rate of the user input but also increases the probability of false positives.

```
 1  <?xml version="1.0" encoding="utf-8" ?>
 2  <Gesture>
 3    <Name>Nudge Action</Name>
 4    <Description>
 5      move finger forth and back on the Z axis
 6    </Description>
 7    <Action>click</Action>
 8    <Threshold>5</Threshold>
 9    <Length>10</Length>
10    <Data>
11      <Hand id="1" type="position">
12        <set>
13          <z>-2.0, -1, -0.7, -0.5, 0,2.0, 1.7,1.5, 1.0, 0.5</z>
14        </set>
          ...
15      </Hand>
16    </Data>
17  </Gesture>
```

Listing 5.2: Gesture Definition Format

Lines 10-16 define data series, which are matched by the DTW algorithm with the user input and are categorized into different sets. Each set must have at least one data series on one axis and is used to distinguish different movement patterns that result in the same gesture. The DTW distance is calculated by calculating the DTW distance of each data series, getting the average of each set and then calculating the average of them. Values used in the sets are comma separated. This contradicts with the XML specification, but keeps the file small and readable for human beings. The `Hand` element (lines 11-15) is used to assign the data sets to a hand, i.e., it is possible that the system detects gestures that are performed with two hands. This feature is not in use, though, because it showed as unpractical since it is difficult for the user to learn the two hand gestures and the user has to stay with his hands in the limited camera range of the Kinect during the gesture.

6

ProViewKinect

The usability research of three dimensional gesture recognition software for manipulating process models is difficult since there are no implementations yet. The vision is to modify process models without the need of any traditional input methods but only using gestures and finger pointing in a three dimensional space. Therefore a prototype has been created to validate the viability of three dimensional process model manipulation.

Section 6.1 explains the architecture of the prototype. Section 6.2 showcases an implementation of the prototype. Section 6.3 validates the presented prototype.

6.1 ProViewKinect Architecture

The focus when designing ProViewKinect is to build its parts flexible and with clean interfaces in order to be reuse them in other projects (e.g., the «REST Library» is being used in other implementations). The Kinect-based process modeling program is split into three basic components (cf. Figure 6.1) and is based on the *Model-View-Presenter*-Design pattern [GHJV94].

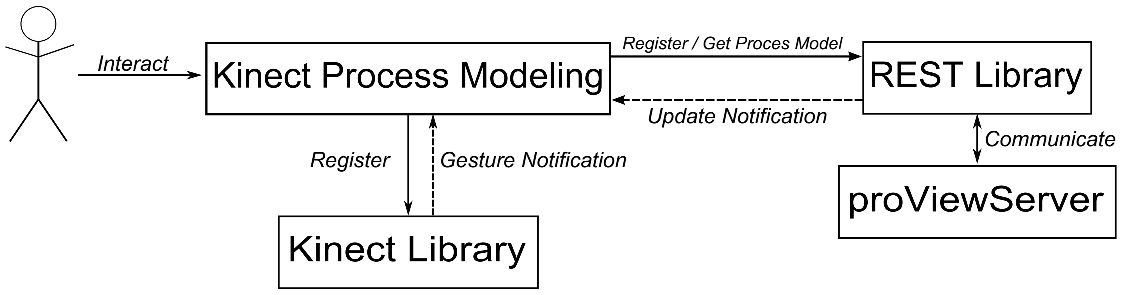


Figure 6.1: The Architecture of the Prototype.

The main component of the architecture is the «Kinect Process Modeling», which provides the graphical user interface (GUI) as well as the application logic and is further described in Section 5. It communicates with the «REST Library» and the «Kinect Library» in order to retrieve the information and updates for the GUI and send requests to the «Kinect Library» when necessary.

The second component is the «REST Library» (cf. Section 3), which is an interface to communicate with the proViewServer (cf. Section 2.2). The GUI can request graphs and change operations with it. Furthermore the «REST Library» can poll the proViewServer (cf. Section 3.3) and inform the GUI about changes to the graphs.

The third and last component is the «Kinect Library» (cf. Section 5) that informs the GUI about gestures and whether the user is pointing with his fingers in order to select a graph element. The «Kinect Library» can be seen as an interface to the official *Microsoft Kinect* SDK [Win13b].

6.2 ProViewKinect Prototype

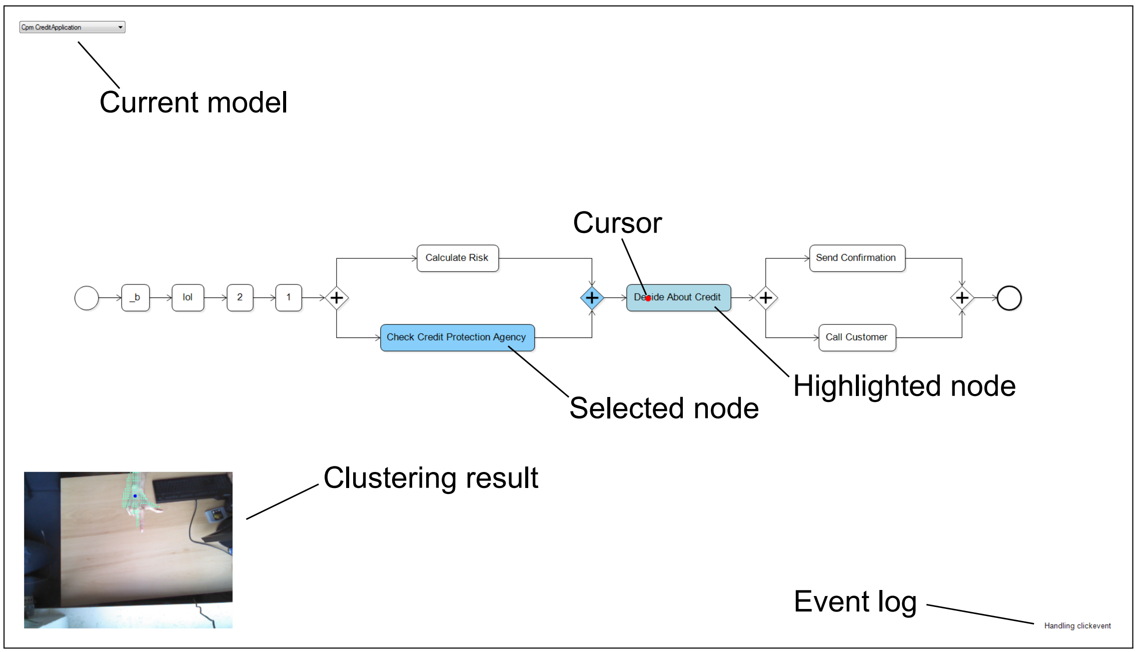
The ProViewKinect prototype, written in C**#**, includes all components of the architecture (i.e., «REST Library»,«Kinect Process Modeling», and «Kinect Library») and features described in the previous sections.

After the prototype is executed, the user is promted to select a display device, on which ProViewKinect's main interface is to be displayed. This is convenient and is required to support beamers that can not be used as main display.

The main interface is divided into three areas (cf. Figure 6.2). In the center, the current process model is presented by using Processing Process Models algorithms (cf. Section 4.1 and 4.2). On the bottom left corner, the result of the CCT NUI's clustering algorithm is shown and serves as feedback whether the user's hands are detected correctly. On the bottom right corner, an event log, used for error messaging, notifications, and debug purposes is displayed. As shown in the bottom left corner of Figure 6.2, the user can use his index finger to move the cursor on the screen. The data for this interaction provided by the «Kinect Library» (cf. Section 5). When the cursor is moved over a node of the process graph, the node is highlighted. To select a highlighted node, the user may show his thumb additionally to the index finger to the Kinect camera. If the highlighted node, as in the figure, was already selected, it gets deselected via versa.

In oder to manipulate the process graph, the user has to show five fingers to the camera. The «Kinect Library» notifies the GUI about the changed finger count and that the graph manipulation mode, requesting a gesture recognition from the «Kinect Library», is enabled. The user is notified by a sound effect and can start making a gesture using his index finger. When a gesture is recognized and the associated action is possible (i.e., the amount of selected nodes is equal to the requirements of the action), the «REST Library» is invoked with the desired action (cf. Section 3.2.3). Should the request succeed, the GUI is notified by the «REST Library» and the process model is updated by adding, removing edges, and moving the nodes to their new positions (cf. Section 3.4). Otherwise, the «REST Library» reports an error and the event log is appended with an error message stating the reason (cf. Section 3.2.4 Unsuccessful Request).

When an external change on the process model gets detected (cf. Section 3.3), only the nodes and edges of the modified part of the graph gets moved. The nodes that are not affected by the change stay in place and the selections are maintained. This was done to avoid an interruption in the users workflow.



Handling clickevent

Figure 6.2: ProViewKinect Graphical User Interface

6.3 Validation of the Prototype

The main question is whether the prototype is accepted by the users. This goes hand in hand with the question how natural, intuitive and exhausting the input interface is.

The pointing gestures used to select and deselect nodes feels very natural and intuitive although it uses the finger position (cf. Section 5.4.3) instead of the pointing vector (cf. Section 5.4.2) to move the cursor. Though the user can brace his elbow on the working surface, the interaction is exhausting. This is partly because the user has to move his fingers in a limited area. Furthermore, it requires some time to get used to concentrating simultaneously on cursor movement and clustering image (providing the detection limits). The fatigue steadily increases the more process nodes the user wants to select at a time. Therefore, a user using traditional input methods can work longer and is faster.

The actual selection by highlighting a node with the index finger and stretching the thumb, to select, is fast and performant. The fatigue is small because the user has only to move his thumb and feels intuitive after being learned.

To manipulate the graph, actions can be performant on selected nodes. To initiate an action, the user may perform a finger gesture. This feature could be very fast and performant, if the system was precise enough to detect the slightest finger movements. Since it is not, the user has to move his hands to perform a finger gesture. While the gestures are small and seldom used, hence, not very exhausting, they feel not intuitive since they are not directly connected with the selected process nodes. It is also difficult to memorize all possible gestures. Supporting the user, a context sensitive interface showing possible gestures could help solving this problem. The gesture recognition system itself is fast enough to keep up with traditional input methods and furthermore feels more natural.

All in all the system is usable and intuitive but exhausting since the finger pointing system is used most of the time. Therefore, it is only recommended for small changes on process models. The gestures are performant but not very intuitive. A user using traditional input systems has still an advantage over a user using the ProViewKinect’s pointing and gesture interface but the latter is more natural and easier to learn. The graphical user interface itself is clean, simple and offers enough feedback to the user.

7 Conclusion

The ProViewKinect has been presented utilizing the Kinect to provide a finger gesture input interface for manipulating process graphs. It has proven the applicability of gesture based process manipulations and serves as a first step in that direction. This interface is designed to provide a natural and intuitive input experience. It is clear, though, that more effort has to be made to guaranty a more fluent workflow to compete with other input methods. Compared to the traditional input method it is still much slower. Solving the resolution problem (cf. Section 5.5 and 6) would surely improve the speed of the system so that it can compete with other input methods and would increase the detection area. At the moment, the main advantage of ProViewKinect are environments where the use of traditional input methods is difficult since only a RGB-D camera, that can be mounted anywhere, is required.

8 Summary and Outlook

First, the three components of the ProViewKinect prototype has been introduced. «REST Library» communicates with the proViewServer to obtain and maintain the process models (cf. Section 3). «Kinect Library» tracks the users hands using the modified CCT NUI library (cf. Section 5.3), which utilizes the Kinect's depth camera (cf. Section 5.2). Furthermore, it provides finger tracking (cf. Section 5.4) and gesture recognition (cf. Section 5.5) by analyzing the depth data stream provided by CCT NUI. «Kinect Process Modeling» serves as GUI for ProViewKinect and comprises algorithms for process model representation and user interaction (cf. Section 4). Second, the ProViewKinect is presented utilizing the three components. With it the user can access and manipulate process graphs using gestures in three dimensions (cf. Section 6).

Several problems have been discussed resulting of the Kinect's low camera resolution (cf. Section 5.5,6). Therefore, the next logical step would be the new version of the Kinect offering HD resolution cameras [Ter13]. There are other interesting developments, too. The *leap motion*, for example, seems to be a precise hand tracking device (developers speak of a sub-millimeter accuracy [lea13]). This detection is limited to a range of 1 meter though [Joe13]. The benefit of the leap motion compared to the Kinect would be that it resolves most of the problems arose from the inaccuracy of the detection. The pointing input method for example (cf. Section 5.4) would most likely be possible with the additional accuracy thus enabling finger gestures without the need of moving the hand.

List of Figures

Listings

Bibliography

[Bür12] BÜRINGER, Stefan: *Development of a Business Process Abstraction Component based on Process Views*. http://dbis.eprints.uni-ulm.de/838/. Version: 2012. – Bachelor Thesis, Uni University

[Cra13] CRAIG EISLER: *Near Mode: What it is (and isn't)*. blogs.msdn.com/b/kinectforwindows/archive/2012/01/20/near-mode-what-it-is-and-isn-t.aspx, 2013. – [Online; accessed 11-July-2013]

[Dap12] DAPPER, Matthias: *Implementation of a Multi-Touch, Gesture-based Process Modeling Component for Apple iPad*. http://dbis.eprints.uni-ulm.de/883/. Version: 2012. – Bachelor Thesis, Uni University

[GHJV94] GRAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph ; VLISSIDES, John: *Design patterns-Elements of Reusable Object-Oriented Software*. 1994

[Joe13] JOE LIMON: *Image Files*. https://forums.leapmotion.com/showthread.php?1005-Technical-specifications&p=5795&viewfull=1#post5795, 2013. – [Online; accessed 14-July-2013]

[Jos13] JOSH SUTPHIN: *Doing Thumbstick Dead Zones Right*. http://www.third-helix.com/2013/04/doing-thumbstick-dead-zones-right/, 2013. – [Online; accessed 14-July-2013]

[KKR12a] KOLB, Jens ; KAMMERER, Klaus ; REICHERT, Manfred: Updatable Process Views for Adapting Large Process Models: The proView Demonstrator. In:

Demo Track of the 10th Int'l Conf on Business Process Management (BPM'12), 2012 (CEUR Workshop Proceedings 940), 6–11

[KKR12b] KOLB, Jens ; KAMMERER, Klaus ; REICHERT, Manfred: Updatable Process Views for User-centered Adaption of Large Process Models. In: *10th Int'l Conference on Service Oriented Computing (ICSOC'12)*, Springer, October 2012 (LNCS 7636), 484–498

[KR13a] KOLB, Jens ; REICHERT, Manfred: Data Flow Abstractions and Adaptations through Updatable Process Views. In: *28th Symposium on Applied Computing (SAC'13), 10th Enterprise Engineering Track (EE'13)*, ACM Press, March 2013, 1447–1453

[KR13b] KOLB, Jens ; REICHERT, Manfred: Supporting Business and IT through Updatable Process Views: The proView Demonstrator. In: *ICSOC'12, Demo Track of the 10th Int'l Conference on Service Oriented Computing*, Springer, March 2013 (LNCS 7759), 460–464

[KRR12] KOLB, Jens ; RUDNER, Benjamin ; REICHERT, Manfred: Towards Gesture-based Process Modeling on Multi-Touch Devices. In: *1st Int'l Workshop on Human-Centric Process-Aware Information Systems (HC-PAIS'12)*, Springer, June 2012 (LNBIP 112), 280–293

[lea13] LEAP MOTION TEAM: *Inside Leap Motion: Meet our Hardware Engineers*. `http://blog.leapmotion.com/post/55366438663/inside-leap-motion-meet-our-hardware-engineers`, 2013. – [Online; accessed 14-July-2013]

[LWZ+09] LIU, Jiayang ; WANG, Zhen ; ZHONG, Lin ; WICKRAMASURIYA, J. ; VASUDEVAN, V.: uWave: Accelerometer-based Personalized Gesture Recognition and Its Applications. In: *Pervasive Computing and Communications, 2009. PerCom 2009. IEEE International Conference on*, 2009, S. 1–9

[MPC12] MINSUN PARK, Jaemyun K. Md. Mehedi Hasan H. Md. Mehedi Hasan ; CHAE, Oksam: Hand Detection and Tracking Using Depth and Color Information. (2012). http://www.academia.edu/1542615/Hand_Detection_and_Tracking_Using_Depth_and_Color_Information

[MR81] MYERS, Cory S. ; RABINER, Lawrence R.: Comparative Study of Several Dynamic Time-Warping Algorithms for Connected-Word Recognition. In: *The Bell System Technical Journal* 60 (1981), Nr. 7

[PSH97] PAVLOVIC, V.I. ; SHARMA, R. ; HUANG, T.S.: Visual interpretation of hand gestures for human-computer interaction: a review. In: *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 19 (1997), Nr. 7, S. 677–695. http://dx.doi.org/10.1109/34.598226. – DOI 10.1109/34.598226. – ISSN 0162–8828

[Rub91] RUBINE, Dean: Specifying gestures by example. In: *SIGGRAPH Comput. Graph.* 25 (1991), Juli, Nr. 4, 329–337. http://dx.doi.org/10.1145/127719.122753. – DOI 10.1145/127719.122753. – ISSN 0097–8930

[Ste13] STEFAN: *Candescent NUI*. http://candescentnui.codeplex.com/, 2013. – [Online; accessed 10-July-2013]

[Ter13] TERRENCE O'BRIEN: *Microsoft's New Kinect is Ffficial: Larger Field of View, HD Camera, Wake with Voice*. http://www.engadget.com/2013/05/21/microsofts-new-kinect-is-official/, 2013. – [Online; accessed 14-July-2013]

[Wik13] WIKIPEDIA: *Quantization (signal processing) — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=Quantization_(signal_processing)&oldid=560924135. Version: 2013. – [Online; accessed 30-July-2013]

[Win13a] WINDOWS TEAM: *Kinect for Windows Sensor Components and Specifications*. http://msdn.microsoft.com/en-us/library/jj131033.aspx, 2013. – [Online; accessed 22-July-2013]

[Win13b] WINDOWS TEAM: *Microsoft Kinect SDK*. http://www.microsoft.com/en-us/kinectforwindows/develop/developer-downloads.aspx, 2013. – [Online; accessed 15-July-2013]

Name: Hayato Hess Matrikelnumber: 698230

Deklaration

I declare that I completed this work on my own and that information which has been directly or indirectly taken from other sources has been noted as such.

Ulm, the ....

Hayato Hess