GRAPHICAL LANGUAGE FOR LIDAR DATA PROCESSING

HIGH LEVEL DESIGN

One of the results of the phase I efforts of this SBIR was to develop a set of detailed requirements to use for the development of the system. This involved meeting with customers and potential customers for the software and gathering information relating to the specific needs.

Given this information, the phase II efforts focused on the design and development of the system. During this time many hours were spent graphing concepts and driving those concepts down to the level of class diagrams for the development efforts. The following sections summarize this work.

The ALPS system consists of the following major components:

- Graph Engine
- Execution Engine
- Algorithm Interface Framework
- Data Handling Framework
- User Interface

Each of these major components will be described in the following sections.

GRAPH LANGUAGE ENGINE

The core component of the ALPS system is the graphical language used to represent the processing workflow definition. I call this a language, as it is truly a development language, albeit a graphical one. This graphical language defines all of the constructs necessary to create extremely complex processing flows for the system. In addition to supporting ALPS, this language could be extended to support other applications, as this problem is not unique to LIDAR processing.

Graphs

The core component of the language is the graph. A graph represents the set of processing steps required to be performed on any given set of data. The graph contains multiple nodes. A node is a single operation to be performed on a set of data. Each node contains zero-or-more input ports, output ports, and parameters. An input port is a point at which data can be connected to flow into a node. Similarly, an output port is a point at which data will be output from a node. Ports are connected via links to each other. Each link connects a single output port, to a single input port. Parameters are used as input to control the processing of the node, and are defined on a per-node basis.

An interesting twist to this simple structure is that a graph is also a node. This simple design construct is what provides the capability to support nested sub-processes within the process flow. Because a graph is a node, a graph can be placed anywhere a node can be placed, thus creating a nested structure. This also means that a graph must also have multiple input ports, output ports, and parameters.

Figure 7.1 is a graphical representation of this structure.

Nodes

There are several types of nodes in the system. The type of node is denoted within the structure by a field called “NodeType”. It is important to note that the system processor really does not care about node type,
it handles all nodes in the same way. Node types are used, however, to load the actual code which will get executed for that node. For example, for “Graph” type nodes (i.e. a node that is a graph) loads the graph processor to execute that node. Algorithm nodes, on the other hand, load the algorithm code to execute the node. The processing system, however, does not distinguish. Each node is handled by loading the code for the node, processing all of the links to the pass data to the node, and executing the node. Output data is then transferred to the output ports to be transferred to other nodes. More specifics on the execution engine will be discussed in later sections. The following node types are recognized by the system at this point (new node types can easily be added later).

Algorithm – The Algorithm node type represents an algorithm to be applied to the data for processing. This is the primary source of processing data. Algorithms can be added and removed from the system quite easily with the algorithm processing engine.

Graph – The Graph node type represents a graph within the system. The graph will include multiple nodes of its own and has the ability to do complex processing within itself, and yet be represented by a single node at upper levels.

Edit – The Edit node provides a facility to allow a data editing step to be added to a processing flow. This allows for human processing to be integrated directly with automated processing.

Review – The Review node provides a mechanism to allow the user to review the results of processing steps. The user can respond positively or negatively to a review. Negative responses cause the processing flow to return to an earlier part of the process.

Data – The data node provides the essential mechanism for integrating data with the system. A data management framework allows data be imported into the ALPS system, and nodes are created from those data sets. That data is then integrated into process. Data nodes can either receive (store) data, or provide (send) data.

Internal – An internal node provide a means where the system has a built-in mechanism for performing some operation on the data. This has similar functionality to the algorithm, but has a more robust interface for working with data and integrating within the process.

To create new types of nodes is a relatively straightforward process. A new Node Type must be created, and the code associated with that new type must be written and integrated within the system. In this way the system can be expanded to include many types of nodes. In general this will not be necessary, as the algorithm node provides a robust functionality for processing data.

Node Catalog

Because the system is anticipated to contain many nodes, a catalog system had to be created for managing this. The node catalog is a component of the Graph Engine. At any given time there are two available catalogs, the project catalog and the system catalog. The project catalog contains all nodes associated with the current project. This will generally include graphs and data nodes, but may include any type of node. The project catalog provides a nice mechanism for an algorithm process engineer to manage his multiple graphs, sub-graphs, and data.

The system catalog is shared across all projects. This catalog is managed by an administrator and generally contains all of the algorithms, internal nodes, and possibly data nodes used across the system. Processing engineers will drag nodes from this catalog and drop them into their graphs to create processing flows.

EXECUTION ENGINE

The ALPS execution engine executes graphs as defined in the previous section. The execution engine is based on a dynamic loading, interpreted, data-driven, message passing paradigm. The execution engine’s starting point is a graph to be executed. The module that does this is the graph execution module. This module loads the internal representation of a graph, and converts that graph into an executable set of node objects called ExecNodes.

ExecNodes are processing objects within the system. Each NodeType is directly associated with exactly one ExecNode object. The system then uses the NodeType field within the node to create objects to perform the actual execution of that node. As an example, the AlgorithmNode type is associated with an AlgorithmExec object. This object is responsible for dynamically loading the appropriate algorithm code and executing it. Each node has another field called NodeName. The NodeName is used by the execution
object to identify the specific item to execute. For algorithms this is a DLL built by the algorithm integration framework. For graphs this is the name of the graph to load for execution. In this way, the NodeName and NodeType fields specifically identify an exact item to execute.

The process begins by sending a message to all nodes within a graph to Initialize. This message allows each node to perform and required initialization steps, such as connecting to databases, etc. The next message sent is a Start message. This message informs each node that it can begin processing immediately. Each node then tests to see if any data is required for processing (i.e. are there any input ports awaiting data). If input data is required, the node likely performs no immediate action. If no input data is required, the ExecNode will execute its specific code. Each node can run in its own thread, allowing for maximum parallelization. In this way, the system is inherently data-driven.

It is important to note here the relevance of a data-driven architecture. A data-driven system provides the absolute maximum parallelization of processing possible. There is no inherent order of operations within the system. The order in which processing steps occur is strictly guided by the availability of data for which to process. The graphical process definition is what defines these complex data dependencies.

When a node completes its processing function, it sends a message containing the resultant data to its appropriate ports. This message is propagated via links to input ports on other nodes. When a node receives an input port data message, that node checks all of its input ports for data. If all ports have received data, that node’s data dependencies have been fulfilled, and it can then perform its processing. When that node completes, it sends its messages and processing continues to the next node.

A big question here is, how can the graph ever execute if all nodes are waiting on input data? This is solved with data nodes. A data node which provides data has only output ports, thus it has no input data dependencies. When the Start message is received by a data node, that node immediately sends data messages to all connected processing nodes. Because of this, a graph requires some type of data node to begin processing. This makes sense, as a graph with no input data cannot function.

**ALGORITHM INTERFACE FRAMEWORK**

The Algorithm Interface Framework component of the system is what allows algorithms to be integrated into the ALPS system. One of the design goals of the system was to provide a simple mechanism to integrate new and existing algorithms into the system. This component is what accomplishes that goal.

The framework consists of the AlgorithmExecNode object within the execution engine, as well as a set of tools and APIs (the Algorithm Developer Toolkit) to facilitate integration. Algorithm code is compiled into Dynamic Link Libraries (DLLs) and registered into the system via configuration files. Algorithm developers use the Algorithm Development Toolkit to build algorithm DLLs, which are then integrated into a system by an administrator.

The AlgorithmExecNode object receives data messages from the execution engine until all data inputs are satisfied. Once all data inputs are satisfied, the object loads the algorithm DLL and executes the algorithm passing all of the required data. The algorithm processes the input data and creates a set of output data. Once complete, the AlgorithmNodeExec object sends data messages to all of its output ports.

The Algorithm Developer Toolkit consists of a set of APIs, MS .NET project files, data integration tools, and testing tools for algorithm developers. Algorithm developer’s primary responsibility is to implement a single interface which has the following format:

```
Process(Dictionary<string, Data> In, Dictionary<string, Data> Out, Dictionary<string, string> Params)
```

This single function interface is all that is required for an algorithm to perform its job. The In data structure is a dictionary of the input data required by the algorithm. Note that a dictionary is a lookup table of multiple items, whereby a string can be used to lookup a specific entry. This dictionary approach allows the algorithm developer to directly ask for required input data by name. As an example, the syntax for accessing input data named “grid” would be In[“grid”]. The same mechanism is used to access the output data and parameters.

The Data object contains all of the functionality necessary to access the data, independently of the actual data format. In this way the algorithm developer is abstracted from the specific data file formats, and can
focus on simply processing the data. The actual data may reside in a database, a flat-file, or any other means necessary.

The Data interface will be further discussed further in the next section.

**DATA HANDLING FRAMEWORK**

The Data Handling Framework is a set of tools and APIs for working with data in the system. The tools allow users to import and export data in various formats as well as manage datasets within the system. The APIs provide a layer of abstraction away from specific data formats, allowing data to process through the system without regard to the specifics of the data format.

The Data Handling Framework manages three distinct types of data; Random Point data, Gridded data, and Vector data. These three types of data encapsulate all of the needs for processing LiDAR data into usable products. All types of data within the system include dereferencing information. The GEOTiff specification is used to represent geospatial references, as this standard is well established and is applicable to all types. LAS for example uses the GEOTiff specification for its georeferencing.

Random Point data is used to represent the actual LiDAR data. LiDAR data is in the form of random points, each with an X, Y, and Z value. The Random Point data supports these values directly, but also provides for additional values to be associated with each point via named properties. This allows for additional information that is often associated with LiDAR data to be represented, such as intensity, date/time, swath number, etc. Random point data can be ingested into the system from either XYZ files, or from LAS files. A framework exists to support any type of data file by implementing certain APIs.

Gridded data is used to represent regularly spaced data such as DEM. This format is useful for many applications of LiDAR data as it is much simpler to work with than random point. Gridded data includes bounding information, spatial resolution information, etc.

Vector data is used to represent vector information which may be produced by the system. This may include Triangular Irregular Network (TIN) data, building detection areas, area selections, etc.

The Data Handling Framework provides the basic capability to pass data around the system without the concern for specific data types. Ports on nodes, however, specify exactly which of the three basic types of data are required for that port, and will not accept the wrong type of data. This ensures that an algorithm which is expecting gridded data never receives random point data and vice versa.

In addition to the Data Handling Framework API, various tools are provided that allow data to be ingested into and exported from the system. Several formats are supported including LAS, NetCDF, (ESRI) Arc ASCII Grid, and (ESRI) TIN. The ESRI specific formats require that an ESRI compatible server be available for storage and manipulation of that data.

**USER INTERFACE**

The ALPS user interface allows the user to build and execute LiDAR processing flows as well as manage data and algorithms. The UI is organized into five panels. On the left side of the user interface are three palettes of Ports and Nodes which can be embedded into a Graph. The center panel displays the graph itself, and the right panel displays properties associated with the currently selected item. This provides an intuitive interface to the user, and mimics the layout of other similar types of software.

The first palette on the left side of the screen is the ports palette. This palette shows the various types of ports which can be added to the graph. This includes random point, gridded, and vector type ports. Ports that are placed on a graph define the interface to the graph from the outside world.

The second palette on the left side of the screen is the current project palette. This displays all of the graphs and nodes associated with the current project. The menu bar includes functionality for creating new nodes and graphs within the current project, and these items will show in the project palette. Projects can be opened and closed via the File menu. This allows the user the ability to manage many graphs and organize those graphs into projects. Each project is associated with a catalog, as described above.

The third palette on the left side of the screen is the system palette. This palette contains all of the nodes installed into the system by the administrator. The user has no ability to add or delete items to/from this palette; that function must be performed by an administrator. This palette will generally contain
algorithms, editing, and reviewing nodes that are currently installed, however it is not limited as such. Datasets or even other graphs can be added to this palette by the system administrator. This allows the administrator to create a shared set of nodes to be used by all users of the system. This may include shared datasets or shared graphs as necessary.

The center of the screen contains the current working graph. The user builds a graph by dragging nodes from the palettes on the left side of the screen, and dragging lines to connect the nodes (via ports). This makes for a very simple user interface for defining processing steps and integrating those steps into a process flow. Nodes are represented by a box which graphically depicts the type of node and its ports and port types. Ports are displayed on the left and right side of the box, with input ports being on the left and output ports on the right. Each port shows the type of data to connect to it via an icon designating random point, gridded or vector data. The user simply connects multiple processing nodes to each other moving logically from left to right to build the graph.

The graph itself has input and output ports and those are displayed as icons designating the type of data that port can accept. The user can add ports to his graph by dragging ports from the ports palette. Ports on the users graph can then be connected to nodes within the graph, thus providing an interface from the world outside the graph to the nodes inside of the graph.

If a node which is placed on the graph is itself a graph, its definition can be viewed by double-clicking on the node within the graph window. This will automatically open the graph associated with that node and allow editing to take place (if the graph is editable). This makes the process of creating and editing nested complex processing flows very simple.