

[0144] Turning now to FIG. 9, a process 600 for generating byte code is shown. The process takes as inputs previously stored application resources (step 602) and obtains a first scene (step 604). The process then obtains a first node in the scene (step 606). Next, the byte code execution statements for the node are retrieved (step 608). This step is illustrated in more detail in FIG. 10. From step 608, the process determines whether additional nodes need to be processed (step 610), and if so, obtains the next node (step 612) before looping back to step 608.

[0145] When all nodes have been processed, the process creates the VcCGMethod and stores statements associated with the scene (step 614). Next, the process determines whether additional scenes remain to be processed (step 616). If so, the next scene is obtained (step 618) and the process loops back to step 606 to continue the byte code generation process. When all scenes have been handled, the process then creates the byte code execution image (step 620) before exiting (step 622).

[0146] Referring now to FIG. 10, a process 630 for obtaining byte code execution statements for an object node is shown. First, the process determines the node type (step 632). If the node is a shape, the process generates a shape creation statement (step 634) as well as a statement to begin property capture (step 636). Steps 638-644 then generate statements to set each retrieved property. Finally, the process creates a statement to commit the properties (step 646).

[0147] On the other hand, if the node is a query data source, the process obtains a query and generates a parameterized SQL statement (step 648). The process then determines column names to be used as referenceable identifiers (step 650). Next, a statement is created to create a query (step 652). Steps 654-660 then create one or more statements which set each query parameter value. Finally, the process creates a statement to execute the query (step 662). From step 646 or step 662, the process exits (step 664).

[0148] Referring now to FIG. 11, a process 666 to execute the byte code is shown. First, an execution context is created (step 668). The process then calls VcCGMethod Performo method (step 670). The statements are retrieved (step 672). Next, the statement is classified (step 674). From step 674, when a VcStmtCreateShape statement is encountered, the process generates a new shape (step 676) and stores the shape in a context-hash table using a tokenized shape name (step 678). From step 674, when a VcStmtBeginProperties statement is encountered, the process looks up the shape information in the hash table (step 680). The process then calls the shape's OnPropertiesBegin() method (step 682). When a VcStmtSetProperty statement is encountered, the process looks up the shape information in the hash table (step 684). It then evaluates the property expression (step 686) before calling the shape's SetProperty() method to assign the value to the property. When a VcStmtEndProperties statement is encountered, the process looks up the shape information in the hash table (step 690). It then calls the shape's OnPropertiesEnd() method (step 692). The shape is then initialized (step 694) and added to a canvas display list (step 696).

[0149] From step 678, 682, 688 or 696, the process then checks whether additional statements need to be executed (step 698) and if so, the next statement is obtained (step 800) before the process loops back to step 674.

[0150] From step 698, the process then deletes the execution context (step 802) and refreshes the canvas (step 804). Finally the process exits (steps 806).

[0151] A chart illustrating a graph editing system is shown in FIG. 12. A scene graph 480 provides a hierarchical representation of an application. Each element of the scene graph 480 is called a node and the node is used in generating a byte code. Additionally, two views 482 and 484 of the data are shown. The view 482 is a layout (a map) showing data that has been retrieved from two datasets.

[0152] In the view 484, a datapoint may be represented as a single image 485, which is highlighted in FIG. 12. The image 485 is shown in the design mode with a placeholder image. Attributes associated with the image 485 are shown in a window 486. The user can edit any data element since each data element has its own drawing window. Thus, a graph may be placed at the datapoint so that a graph exists within a data point.

[0153] Since the editing system of FIG. 12 allows the user to define graphs within data points that in turn are nested within graphs themselves, the editing system of FIG. 12 provides a way to drill down and see more detail. Thus, when used to show scenes with varying levels of detail, the user may zoom into a dot which turns into a graph and, when the user zooms into the graph, its datapoints turn into additional graphs to allow the user to drill down for more detail. The graph editing system provides a convenient way of editing the representation of a single datapoint and any arbitrary representation for a datapoint that may be as general as the parent scene itself.

[0154] The mathematics of a wormhole is discussed next. Wormholes are special objects that allow a user to look through a window in one scene to another scene. FIG. 13 illustrates this effect and shows how a composite zoom factor is computed for the target scene.

[0155] FIG. 13 shows a wormhole, which is a type of hyperlink that allows the user to pass context information through the hyperlink and at the same time see through the hyperlink to the other side. Since the other side of the hyperlink is observable, that side is "transparent" to the user. In FIG. 13, a wormhole 490, shown as a window from a first scene 492 to a second scene 494. The wormhole 490 provides state information, as opposed to a conventional hyperlink which is stateless. From a user location 491, relationships between a first elevation (elev₁) 496 to the first scene 492 and a second elevation (elev₂) 498 to the second scene 494 may be expressed as:

$$elev_1 = \frac{1}{Zoom_{scene1}}$$

$$elev_2 = \frac{1}{Zoom_{scene1}} + \left[\frac{1}{Zoom_{wormhole}} - 1 \right]$$

[0156] In this case, each of the Zoom factors represents a magnification factor. Moreover, the Zoom factors are the reciprocal values for the associated elevation parameters. For instance, at a 100% Zoom factor, the elevation parameter is 1, while at a 200% Zoom factor, the elevation parameter is 0.5.