VAPORGUI Internals Refactoring

Control Executive proposal

Overview & background

One component of the refactoring plan is to provide a high level "control executive". By this is meant an API to control most of the operations that are currently under the control of the VAPOR GUI, while avoiding any actual GUI code. Here are some thoughts about what should be doable from the Control Executive:

- Load and Save Session [currently in the Session class]
- Create and destroy visualizers [currently performed by the VizWinMgr class]. Note: In the absence of the GUI or windows, a visualizer should be replaced by something that encapsulates a rendering context
- Create and destroy renderer instances [currently performed in the event routers in the GUI]
- Create and destroy Params instances [currently performed in the Params class]
- Get and set the state of any Params instance.
- Perform a rendering
- Play and record an animation sequence

The above capabilities primarily are currently in the following classes:

- Params - This class (and its subclasses, in the params lib) is responsible for managing the state of all the renderers. The Params class maintains a database of all the available Params instances for all of the renderer instances. There are also three Params classes (animation, region, and viewpoint) that do not render.
- ParamNode - this enables getting and setting the state of various elements in the Params.
- VizFeatureParams - This is in the GUI; it is not a Params subclass (probably it should be). It contains most of the rendering settings that are not part of the state of the various renderers, such as scene stretching, enablement of color bars, axis annotation settings, etc.
- DataStatus - This class (in the params lib) provides a summary of the state of the data that is currently loaded. It maintains information about what variables exist, what timesteps, refinement levels, and LOD's are present for each of the available variables.
- GLWindow - This class (in the render lib) is a Qt OpenGL Widget, and it also controls the process of rendering images. It sets up the GL state and calls the various renderers at render time. The GLWindow also controls the rendering of auxiliary objects such as color scales. Various "dirty bits" are in the GLWindow class. These are switches that are set when state changes occur that will necessitate a re-render.
- Manipulators - these are currently in the render lib. They are drawn by the GLWindow, but they are controlled by user actions.
- Renderer - These classes are in the render lib. There is a subclass of the Renderer class for each of the Params classes that perform rendering. Instances of these classes are invoked by the GLWindow at render time (by their paintGL method).
- VizWinMgr - This class (in the GUI) maintains information about all the params and the renderers that are available, as well as windows they are using. It creates and destroys visualizers, and makes the appropriate connections between the visualizers and the renderers that operate on them
- EventRouter - These are in the GUI. There is an EventRouter sub-class for each of the parameter tabs in the GUI. These classes keep the Params instances and the GUI synchronized, and have a lot of code for responding in various ways to user actions.
- Session - This class (in the GUI) is primarily concerned with saving and restoring session files. It also handles the reinitialization of the application when a new dataset is loaded.
- UserPreferences - This class is in the GUI. Most of the information in the user preferences panel is only relevant for interactive (gui-based) sessions; however it does contain some state that must be accessed during normal operation, such as the background color, or the name of the log file.
- AnimationController - this class (in the GUI) is responsible for causing rendering to occur and updating the time step whenever the user starts or stops playing an animation sequence.
- Command - This class is in the GUI. Each Command instance records a change in the state resulting from an action in the GUI. If the state change is a change of a params, the command has an instance of the params before and after the change. There are other commands that are not reflected in the params, such as the enablement of a renderer, or the change of tab.

Proposed Control Executive Design:

First, split the above classes into parts that can remain in the GUI and parts that should be represented in the Control Executive.

- State management: The Params class already presents an API (the extensibility API) that is appropriate for the Control Executive. The VizFeaturesParams should be reimplemented as a Params instance. The DataStatus class should me moved to the vdf lib. The Gui-independent parts of the Session class should be moved into the ControlExecutive class, but the command queue should be kept in the GUI. The settings in the UserPreferences that control rendering should be moved to the VizFeatureParams; the rest of it would remain in the GUI.
- Render control: The Renderer classes are already fairly independent of the GUI. The GLWindow class needs to have all Qt dependencies removed. . The drawing of text requires creation of a Qt-independent conversion from text to images. The manipulators should have their interactive control moved to the GUI app, but their drawing should remain in the Renderer lib.
- The VizWinMgr class requires a lot of restructuring. It can mostly be moved into the Control Executive. It also needs extensive trimming to produce a clean API. However, the ability to create and control visualizer windows and the controlling of tabs would remain in the GUI.
- The animationController can remain in the GUI. However the Render lib will need a simple mechanism for starting and continuing animation sequences.
- The Command class can remain in the GUI.

Once the functionality has been split as above, the next step is to clean up and organize the Control Executive API to facilitate development of applications that will use the API. This organization should consist of three sections: State management (the Params API) , Rendering (Render, GLWindow, Manip methods that perform rendering), and ControlExecutive (including VizWinMgr and Session)

Control Executive API

The proposed API is available here. Doxygen source for the documentation is attached here.

(AN) The above API has been modified here with some additional methods and some changes to how Params and Renderers are referenced. Doxygen source is attached here.

The following items are not currently addressed by the ControlExecutive class:

1. ErrorHandling: All of the ControlExecutive methods return a success or failure status. However, no methods have been provided for returning an error code or error message. ## (AN) I propose an ErrorHandler class be defined, which UI's can use to be notified when the Control Executive has an error condition.
2. OpenGL state mgt: In the proposed API the UI is responsible for establishing the OpenGL drawing context (e.g. creating a drawing window) on behalf of the ControlExecutive. We need to flesh out what other OpenGL responsibilities/expectations the UI and ControlExecutive have wrt OpenGL. Not sure what you are asking. Does this require an implementation?
3. Timestep specification: How are time steps specified?## The timestep is specified in the AnimationParams class
4. Key framing:## This is also specified in the AnimationParams class
5. Manipulators## (AN) I propose a ManipParams class that will include all the settings associated with the current mouse mode. UI's can translate mouse events into changes in the ManipParams that applies to that window.
6. Image capture## (AN) I added an "EnableCapture" method to the API doc
7. Probing: How does the UI query data values selected by the probe?## (AN) The UI can convert the probe mouse coordinates into user coordinates (using Extents etc. obtained from DataMgr). Then it can use the RegularGrid to evaluate a variable at the specified position.

Example Use Cases

Case 1

```
// A single visualization window with a single renderer, single instance

ControlExecutive *cntrl_exec = new ControlExecutive();

// The UI has created an OpenGL window. It now informs the ControlExecutive
// that it is available and ready for rendering

int viz = cntrl_exec->NewVisualizer(oglinfo);

// Tell the CE which files to load. LoadData() returns a pointer to a class
// object that can be used to query metadata info
//

const DataInfo *data_info = cntrl_exec->LoadData(file(s));

// Create a new DVR params bound to the single OpenGL window we have
//

// Note that it is adjusted to

// ensure the settings are compatible with the data that has been loaded.

DVRParams* rp = (DvrParams*)cntrl_exec->NewParams("DVR",viz);

//Create a renderer instance suitable for the DVR Params

int inst = ctrl_exec->NewRenderer(viz, "DVR", rp);

// Get a handle for renderer instances Params object and then change
// some parameters. SHOULD CHECK ERROR STATUS
// when setting params. If error occurs on a single SetValue(),

//the value will not be changed and an error return code will be returned.

rp->SetVariable("enstrophy");

rp->SetTransferFunc(mymap);

// Make sure the dvr instance is active and tell the CE to enable
// this renderer, and render everything.

cntrl_exec->ActivateRender(viz, "DVR", inst, true);

cntrl_exec->Paint(viz);
```

Case 2

```
// Here we have  two iso renderer instances, both bound to the same window

ControlExecutive *cntrl_exec = new ControlExecutive();

int viz = cntrl_exec->NewVisualizer(oglinfo);

const DataInfo *data_info = cntrl->LoadData(file(s));
```

```
// Create two iso Params instances

IsoParams* rp1 = (IsoParams*)cntrl_exec->NewParams("Iso",viz);

IsoParams* rp2 = (IsoParams*)cntrl_exec->NewParams("Iso",viz);

// Set some params on rp1 and rp2.  Put these changes as one entry in the command queue, so

//we can undo them if there's an error.  Note that if we did not invoke StartCommand(), then each Params

//state change would be inserted as a separate entry in the queue

ctrl_exec->StartCommand(rp1, "Iso setup");

rp1->SetVariable("u");
rp1->SetIsoValue(99.0);
rp1->SetMappedVariable("enstrophy");
rp1->SetTransferFunc(mymap);

int rc = ctrl_exec->EndCommand(rp1);

// if rc is nonzero, an error occurred and state was reverted to prior to the StartCommand()

if (rc){

  //provide an error message here

}

ctrl_exec->StartCommand(rp2, "Iso setup");

rp2->SetVariable("v");
rp2->SetIsoValue(99.0);

rc = ctrl_exec->EndCommand(rp2);

if (rc){

  //provide an error message here

}

// obtain renderer instances for the Iso Params

int inst1 = ctrl_exec->NewRenderer(viz, "Iso", rp1);

int inst2 = ctrl_exec->NewRenderer(viz, "Iso", rp2);

// Make sure both iso's are active and tell them to render
cntrl_exec->ActivateRender(viz,"Iso",inst1, true);
cntrl_exec->ActivateRender(viz,"Iso",inst2,true);
cntrl_exec->Paint(viz);
```

Case 3

```
// Here we have two visualizer windows with one iso renderer per window

ControlExecutive *cntrl_exec = new ControlExecutive();

// Create two windows
//
int viz1 = cntrl_exec->NewVisualizer(oglinfo);
int viz2 = cntrl_exec->NewVisualizer(oglinfo);

const DataInfo *data_info = cntrl->LoadData(file(s));

// Create one iso instance for each of the two windows
//

IsoParams* rp1 = (IsoParams*)cntrl_exec->NewParams("Iso",viz1);

IsoParams* rp2 = (IsoParams*)cntrl_exec->NewParams("Iso",viz2);

// Set iso params
//
rp1->SetVariable("u");
rp1->SetIsoValue(99.0);
rp1->SetMappedVariable("enstrophy");
rp1->SetTransferFunc(mymap);
```

```
rp2->SetVariable("v");

rp2->SetIsoValue(99.0);

//Create renderers

int inst1 = ctrl_exec->NewRenderer(viz1, "Iso", rp1);

int inst2 = ctrl_exec->NewRenderer(viz2, "Iso", rp2);

// Activate renderers and call Paint for *both* windows
//

cntrl_exec->ActivateRender(viz1,"Iso",inst1, true);
cntrl_exec->ActivateRender(viz2,"Iso",inst2,true);

cntrl_exec->Paint(viz1);
cntrl_exec->Paint(viz2);

// Change viewing params and re-render. Both windows use global view params

//Push and pop on the matrix stack is only performed during rendering, not here
//Note that the visualizer params are by default global; i.e. apply to both viz1 and viz2

//The following two lines are not needed except when the viewpoint params are not already in "global" state

cntrl_exec->GetLocalParams(viz1, "Viewpoint")->SetGlobal(true);

cntrl_exec->GetLocalParams(viz2, "Viewpoint")->SetGlobal(true);

ViewpointParams* vpp = (ViewpointParams*) cntrl_exec->GetGlobalParams("Viewpoint");

Viewpoint* vp = vpp->GetViewpoint();

vp->SetCameraPosition(3.,4.,5.);

vp->SetViewDir(3.,3.,3.);

cntrl_exec->Paint(viz1);
cntrl_exec->Paint(viz2);
```

Case 4

```
// Handing an undo

// The UI should check to find out which params instance
// has changed, and then update its state to reflect change.
// identify the window, instance, type that changed

int instance, viz;

string type;

Params* p = cntrl_exec->Undo(&instance,&viz,type);

//The UI will need to insert code here to update its state for (instance, viz, type)

ctrl_exec->Paint(viz);

//Note: changes that occur always can be identified as a modification of a single params instance.
```

Prototype Implementation proposal

Goal: A simple prototype that validates the Control Executive API, that can be used as a framework for filling in full application capabilities. The prototype will eliminate all legacy code that we are not intending to continue supporting in 3.0.

Implementation plan overview:

1. Create a 3.0 branch from the 2.2.4 tree
2. Leave vdf and common libraries intact (for now)
3. Implement Control Executive as a class (initially) in the Render lib. Hook it up to Params and Render libraries (after following changes are implemented).
4. In Params lib, remove most of the code (i.e. at least remove it from the build) except for the following changes:
    a. Keep Params, RenderParams, ParamsBase classes but strip out legacy code.
    b. Reimplement RegionParams, ViewpointParams, AnimationParams based on XML representation
    c. Keep Box, ArrowParams (but strip out or replace legacy code)
    d. Add Undo/Redo queue management to Params class
    e. Add new Params classes for management of state that is currently not in Params, such as Manips, renderer enablement, instance management.

    f. Load/save session files
    g. Add error checking, reversion of invalid settings on SetValues
    h. Avoid DataStatus usage (rely on DataMgr for VDC information; any necessary functions should eventually be implemented in VDF lib.
5. In Render lib:
    a. Retain Renderer class, plus ArrowRenderer. Clean out legacy code.
    b. Retain Manip class
    c. Strip Qt Dependence out of GLWindow class. Retain ability to maintain Renderer instances via OpenGL calls.
    d. Eliminate dependence on Trackball: The trackball will be in the GUI and will act through the ViewpointParams.
6. Implement a Qt-based GUI
    a. Clean up the EventRouters that correspond to Params classes that are being used in this prototype, retain only functionality that is required for setting values in params and connecting them to GUI events.
    b. Redesign current EventRouter event model, simplify and avoid spurious events.
    c. Replace VizWin with a stand-alone QGLWidget
    d. Purge legacy code from VizWinMgr. Identify (and reimplement) the functionality of VizWinMgr that is needed in GUI (some functionality such as start-up configuration should be moved to Params).