VSim Documentation

The software, VSim, makes use of the following technologies:

- **OpenSceneGraph – 3.4.0**

  OpenSceneGraph is an open source 3D graphics application programming interface.

  Refer to the printed notes for details on what it is, explanation of some common classes and functions provided by OSG and how to implement them.

- **Wxwidgets – 2.2.3**

  wxWidgets (formerly wxWindows) is a widget toolkit and tools library for creating graphical user interfaces (GUIs) for cross-platform applications.

- Visual Studio – 2013
- Windows 7 operating system

File Structure:

Vsim – contains the actual source code organized into different sections as per their functionality.

Vsim dependencies – contains all the stuff that the source code depends on. This means that the open scene graph and wxwidget libraries go here. They have to be built first before they can be used. For building wxwidgets and OpenSceneGraph, refer to Scott's notes.

Vsim -> nrts -> src -> VSim

- Linux
- Menu
- Narrative
- osgNewWidget
- osx
- overlay
- resources
- ui
- windows
- Miscellaneous files

1. **Linux**
   Contains the DroidSans-Bold.ttf and DroidSans.ttf files. These are the truetype font files that are being used by the software to render the text. If you are trying to build the software from scratch, these fonts have to be installed into the system before they can be used.

2. **Menu**
    a. AboutVSimManager
    Contains the function **showDialog()** and **closeDialog()** that is called from VSimApp and basically just creates and destroys the corresponding dialog box.

    b. CCoordinateManager
    Contains the function showDialog() and closeDialog() that is called from VSimApp and basically just creates and destroys the corresponding dialog box.

    c. DisplayManager
    Creates a new object called as m_display_dialog. Now this object calls the DisplayDialog.cpp class from the UI section. It includes showDialog() and closeDialog(). Apart from that it also has something called as updateDialog() that allows the user to see the changes after he has updated the settings. Therefore, g_settings (object created in the settings class) is used to call different functions from the settings class

    d. LightingManager
    Creates an object m_lighting_dialog that calls the LightingDialog.cpp class from the UI section. It has just the showDialog() and closeDialog() function.

    e. ModelInfoManager
    First performs a check to see if a model is loaded into the software, if it is then an object m_model_info_dialog is created that calls ModelInfoDialog from the UI section and creates the appropriate dialog box with the text information.

    f. NavigationManager
    Creates two objects m_settings_gen_dialog and m_settings_adv_dialog that allow calls to SettingsGeneralDialog.cpp and SettingsAdvancedDialog.cpp respectively. Right now the code to update the advanced settings is not written. Otherwise, updateDialog() allows user to see changes that he makes to the settings that fall in the general navigation settings dialog box.

    g. RestrictionsManager
    Creates two objects m_restrictions_dialog and m_restrictions_cust_dialog that makes calls to RestrictionsDialog.cpp and RestrictionsCustomDialog.cpp respectively. Called when locking has been enabled; it allows/disallows users to perform some functionalities.

    h. SaveImageManagergenImage()
    This class contains a function called as genImage(), which takes the filename as an input parameter. It contains the logic for saving the selected image onto the given filename. It does so by creating objects of the classes osgViewer::ScreenCaptureHandler() and CaptureImage().

i. SaveMovieManager
This file contains the logic for creating a movie and saving it onto memory at a certain file location. The recording of the movie begins with startRecord() and ends with endRecord(). The genMovie() function contains the logic for saving the recorded movie and making it executable.

On a side note, what are object wrappers and serializers?

A wrapper completely and correctly records all necessary properties of a scene object and its proto. In the reading process, the wrapper will create a clone from the proto, reread properties and add it to the proper position in the scene graph.

A number of serializers in order are used to implement setting/getting methods of each recordable property. The class inheritance is provided to make sure that members of parent classes are also recorded.

j. Settings

Basically it allows you to set everything that is available in the settings tab. Settings dialog box – set all of that

Settings dialog is divided into three sections: Navigation Settings, Content Creator and Font Colors.

- Navigation settings – allows you to set different values for Field of View, Eye Height, Near clip, Far clip, LOD scale, Acceleration factor, and Rendering. By default, FOV is set to 55, Eye height to 1.5, Near clip to 1 and Far clip to 5000 (note that none of them have a specific unit of measure, and so let us just say that they are "x UNITS"). The acceleration factor is set to 6 by default.

- Content creator – ContentCreator.cpp is called for this.

- Font colors – This setting allows you to set the font, size, color, opacity and shadow for Head1, Head2, Body and Label type of text.

3. **Narrative** -
This section completely deals with handling narratives: adding new nodes, traversing across different nodes

Narrative nodes -> Narrative list -> Narrative player

Each narrative that is created has a reference associated with it, and is obtained by the class NarrativeReference.cpp

NarrativeList is basically a linked list of narrative objects. It allows to add or remove these objects from the list. Each time a new node is added, the selection window is set to that node. Similarly, each time it is removed from the list, the selection window is set to the new node that replaces the deleted node's position in the list.

NarrativeListEditor encompasses the code for the entire narrative toolbar. This is the file which will make function calls to other classes everytime someone tries to add new narratives or remove narratives, transit from one narrative node to another, open an already existing narrative, etc.

NarrativeManager contains the code for performing all the operations that NarrativeListEditor performs, but at a low-level i.e. it attempts to make actual changes to the structure of the scene graph into consideration. On the other hand, NarrativeListEditor makes more of high-level changes.

NarrativePlayer is responsible for performing all the actions that are required to trigger a narrative into the 'play' state as well as whatever happens on the screen while it is in that state. This means the frame handling part is taken care of in this file.

NarrativePlayerEventHandler -> binds the key 'p' to switch between pause and play while the narrative is in the playing state.

4. **Osgnewwidget** –

This is where all the OpenSceneGraph concepts are implemented. So these low-level files are then used by other high-level files based on the application.

Mostly these files deal with building and manipulating the structure of the scene graph that we are viewing. (Refer pg 13 chapter 1 of OSG_notes) A scene graph is basically a hierarchical tree data structure that organizes spatial data for efficient rendering.

So a scene graph tree is headed by a top-level root node. Beneath the root node, group nodes organize geometry and the rendering state that controls their appearance. Root nodes and group nodes can have zero or more children. At the bottom-most level of the scene graph, leaf nodes contain the actual geometry that make up the objects in the scene.

Every application that needs to handle the scene graph typically makes use of the group nodes to organize and arrange geometry in a scene.

a. Widget.cpp

- Most fundamentally the geometry of a scene is dealt with in the following manner:
    1. A new osg::Geometry object is instantiated. In this file, it is m_bgGeometry and m_borderGeometry.
    2. Arrays of vertices, normal, and color data are created and initialized.
    3. All the arrays are added to the Geometry object. Also, an osg::DrawArrays object is also added in order to specify how to draw the data.
    4. Finally, an osg::Geode scene graph node is instantiated and the Geometry object is added to it.
- It is a normal practice to make use of osg::Vec3 for storing vertex and normal information since it is an array of three floating point numbers. Similarly, osg::Vec4 is used for storing color data and osg::Vec2 is used for 2D texture coordinates.
- setVertexArray(), setNormalArray(), and setColorArray() are methods that are used by the application to specify arrays of vertex, normal, and color data. The first two methods takes a pointer to Vec3Array as a parameter whereas the third method takes a pointer to Vec4Array.
- setColorBinding() and setNormalBinding() tell the Geometry how to apply the color and normal data. Over here, the binding is set to osg::Geometry::BIND_OVERALL which means that the single color or normal has to be applied to the entire Geometry.
- addPrimitiveSet() tells the Geometry how to render its data . That is, it allows the application to specify how OSG should draw the geometric data stored in the Geometry object.
- Any geometry that is rendered by the application must be attached to a Geode. Geode provides addDrawable() method to allow the application to attach geometry

The rest of the code deals with mostly setting or resetting some or the property of the scene graph.

b. Canvas.cpp

Now that the Geometry is set in Widget.cpp, we have to bind that Geometry to the OSG leaf node called as osg::Geode so that the required geometry can be rendered. This is the file where relationships between different nodes in the geometry are established. One way to establish these relationships is to use the OSG's group node, osg::Group, that allows the application to add any number of child nodes, which in turn can be also Group nodes with their own children

- In this code, it tries to add different children (objects of the class Widget.cpp) to the Group node using addChild() method. Each time a new child is added, its parent is also set using setParent(). Similarly, there is also a provision to remove children using removeChild()

- Apart from this, there are different methods that allow the application to perform operations on the individual child/parent by making calls to the Widget.cpp class. Such methods include setPosition, setSize, setZOffset, etc.

c. Label.cpp

This file deals totally with laying the groundwork for displaying textual information and makes use of osgText for doing the same. The application uses osgText in the following way:

- To display multiple text strings using the same font, create a single Font object that you can share between all Text objects.
- For each text string to display, create a Text object. Over here it is m_text. Different methods called in order to manipulate and specify stuff like alignment, orientation, position, size, color, etc. Here the Font object that had been created is assigned to the Text object.
- Now all these Text objects are added to a Geode using the same way we did for vertices and colors i.e. by using addDrawable(). Then this Geode object is added as a child node in the scene graph.

d. SerializerCanvas.cpp
e. SerializerLabel.cpp
f. SerializerVSCanvas.cpp
g. SerializerVSLabel.cpp
h. SerializerVSWidget.cpp

- As much as I read up online, serialization in this context deals with writing objects into the memory buffer, enabling simple input/output interfaces to be utilized.
  (Read page 3 for more info on serializers and object wrappers)

i. VSWidget.cpp

This file deals with the actual rendering and placement of objects onto the screen. So basically what it does is, it uses whatever settings that the user has made when the screen just opens up, and calls the Widget.cpp, Label.cpp and Canvas.cpp files to create and place the different objects based on the given settings

5. Overlay

a. BrandingManager.cpp
b. OverlayManager.cpp
c. TextEditDialog.cpp

(Intentionally left blank because I could not understand the purpose behind the Overlay files.)

6. Resources

a. EResourcesCategory.cpp
Every embedded resource category has to be annotated by a different color. The EResourcesCategory.cpp class file does just that. It assigns the colors ‘red’,’green’ or ‘blue’ depending on the type of category a certain embedded resource falls into.

b. EResourcesFilter.cpp
VSim allows us to filter out the embedded resources by ‘category’, ‘title’ and ‘file type’. In doing so, it changes the positions of the embedded resource nodes and this file contains the logic for doing exactly that.

c. EResourcesInfoCard.cpp
This file consists of the code for displaying the information about a certain selected embedded resource every time it is hovered upon. The info card has a set size but comes with a ‘hide’ and ‘expand’ button.

d. EResourcesList.cpp
Just like how it was in Narratives, here, the embedded resources is actually a linked list of the embedded resources nodes. So this file contains the code to add or remove nodes from the list.

e. EResourcesListEditor.cpp
This class encompasses the code for calling several different embedded resource classes that allows us to create an individual node and add/remove it to/from the embedded resources list, display information through info cards each time you hover over a resource, allow you to filter out the resources based on category, title or file type, etc.

f. EResourcesManager.cpp
The EResourcesListEditor.cpp file is called from here, kind of like a superset of the list editor.

g. EResourcesNode.cpp
This file allows the creation of a single embedded resource node, and initializes several parameters for the node such as name, file type, file path, description, the category it belongs to, its reposition value, etc.

h. SerializerEResourcesList.cpp
i. SerializerEResourcesNode.cpp

7. UI

Makes use of wxwidgets for creating the graphical user interfaces. Will be helpful to read up wxwidgets documentation to learn more about the different available functions

a. DisplayDialog.cpp
It uses different functions provided by wxwidgets toolkit to actually create the dialog boxes and all the elements within it, right from sliders to text boxes to checkboxes, etc.

b. ModelInfoDialog.cpp
Creates the About -> Model Information dialog box structure.

c. SettingsAdvancedDialog.cpp
Creates Settings -> Navigation settings -> Advanced settings dialog structure

d. SettingsGeneralDialog.cpp
Creates Settings -> Navigation settings dialog structure

e. RestrictionsCustomDialog.cpp
This particular file deals with some changes that take place once lock has been enabled on the models.

f. RestrictionsDialog.cpp
Same as above

g. LightingDialog.cpp
This class deals with providing different lighting settings such as switching on or off between Ambient Lighting and No Lighting, selecting the appropriate time of the day to be used in the program, the intensity of the shadow, the coordinates of the sun's origin, etc.

8. OSX
Contains certain MAC OS specific files.

9. Windows
Contains the VSim solution files as well as VSim.vcproj files which are the starting point for building VSim each time.

Miscellaneous files

NRTSViewer.cpp

Creates a Viewer object g_viewer, onto which a scene graph will be attached and then rendered. It calls an event traversal method of the Viewer class.

In order to render the view, we have to point a camera towards the scene. By default, Viewer:run() will create osgGA::TrackballManipulator in order to control the Camera.

To set your own manipulator, call Viewer::setCameraManipulator(). In this case, there is no need to write Viewer::run(); instead you have to iteratively update the view changes per frame.

NRTSKeySwitchMatrixManipulator.h -> m_viewer <- VSimApp.cpp

VSimApp.cpp -> main crux of the code base

- ➔ Creates objects of each of the menu classes in the default constructor
- ➔ InitViewer
  - sets up several arguments for the viewer and then sets the camera manipulators.
  - A call is made to NRTSkeySwitchMatrixManipulator using the viewer object m_viewer in order to set the manipulators.
  - Using 3 manipulators here – NRTS (NRTSManipulator()) , USIM (USIMManipulator()) , Terrain (ObjectManipulator()) *All these are custom-made*.
  - Finally, set the CameraManipulator to be from one selected by the KeySwitchManipulator.
  - Set up something called as state manipulators (I believe this is not being used at the moment)
  - A few handlers are initialized and set to their appropriate values. These include handlers that influence how the window size changes, printing statistics on the screen, using the LOD scale to increase and decrease something, etc.
  - In the end, the camera matrices are set by using the setProjectionMatrixAsPerspective with parameters such as fovy?, zNear?, zFar? And aspectRatio
- ➔ There are functions to either retrieve or set the values of fovy, zNear, zFar and aspectRatio

Notes about Camera and viewing:

Initial camera frame is centred at the origin with the view direction aligned with the negative Z axis of the world frame. Theefore, we apply transformations such as translation or rotation to it in order to move the camera frame relative to the world frame i.e. in order to get it to “classical viewing”. From thereon, each distance is computed relative from the viewer to the object

Refer: http://info.ee.surrey.ac.uk/Teaching/Courses/eem.cgi/lectures_pdf/opengl3.pdf

Zfar, znear are distances to plane from the centre of projection; projection plane is orthogonal to the z-axis. Fovy is the angle between the top and bottom planes and is computed as

$$fovy = 2\tan^{-1}\left(\frac{h}{2d}\right)$$

$$aspect = \frac{w}{h}$$

In VSim, the default setup is a perspective projection to view objects within a 55 degrees field of view at a distance of 1.0 (ZNear) to 5000.0 (ZFar) units with an aspect ratio of 16.0/10.0