Overview

Solution code needs to be added to the following files. See the main assignment document for instructions.

1. KochPyramid.py
2. LSystem.py
3. HermiteCurve.py
4. BicubicPatch.py
5. GardenScene.py

Other files included in the starter code are:

1. ModelViewWindow.py: Implements all the necessary code for Model, View (camera) and Glut based windowing.
2. FlythroughCamera.py: Extends View and implements flythrough control. This camera is used in GardenScene.py Move the camera to a position and orientation and press 'p'/'P' to add a control point for the Hermite curve. Once finished placing control points press 'g'/'G' to animate. Press 'c'/'C' to clear the control points. **Note that the "Garden Scene" window needs to be active when using the keyboard control. This is because only that window uses a fly through camera.**
3. GeomTransform.py: Implements geometric transformations such as translate and rotation about an arbitrary axis. These transformations are mainly used by the `View` and `FlythroughCamera` classes.
4. GLDrawHelper.py: Implements some common OpenGL drawing routines. The most commonly used functions from this file used in this assignment are: `drawCheckerBoard`, `draw3DCoordinateAxes`, and `draw3DCoordinateAxesQuadrics`.

Unit test files There are some basic unit test files provided for `HermiteCurve.py` and `BicubicPatch.py` in `HermiteCurveTest.py` and `BicubicPatchTest.py` respectively. These files can be simply executed from the IDE or from the terminal.

```
python HermiteCurveTest.py
python BicubicPatchTest.py
```

User Interface Interactions

All windows provide the same interactions as in Assignment 1 and more. As before `GLUTWindow` handles all the `glut` based windowing functionalities. In addition to that, now we have a `View` class that handles all the view/camera transformations. The `View` class implements a general camera that can rotate and translate in its local coordinate frame. It can also perform Arcball/Trackball rotations and zooming (using the Middle/Scroll mouse button, or CTRL+arrow keys). The `FlythroughCamera` class used in `GardenScene.py` extends the `View` class and provides some additional controls for setting up the Hermite curve and fly through animation.

All windows support the following controls. Note that for keyboard interactions the characters are case-insensitive. Mouse buttons are represented as L/M/R BD or Left/Middle (or Scroll Button)/Right mouse Button Down.

A/D or RBD left/right	Translate camera left/right of lookat
W/S or RBD up/down	Translate camera forward/backward along the lookat
R/F	Translate camera up/down along the up vector
Z/X	Roll camera (i.e. rotate about lookat) left/right
$\leftarrow$/ $\rightarrow$ or LBD left/right	Rotate left/right about the up vector
$\uparrow$/ $\downarrow$ or LBD up/down	Rotate about the (`lookat` $\times$ `up`) vector
Ctrl + ($\leftarrow$/ $\rightarrow$) or MBD left/right	Arcball rotate left/right
Ctrl + ($\uparrow$/ $\downarrow$) or MBD up/down	Arcball rotate up/down
Mouse Scroll up/down	Move model farther/closer
'p'/'P' (GardenScene.py)	Add a point and corresponding tangent for fly through camera.
'c'/'C' (GardenScene.py)	Clear the fly through camera path
'g'/'G' (GardenScene.py)	Animate fly through

Important Note : The rest of this document provides some design and implementation details. It is not necessary to know or understand them fully to complete this assignment. These details are provided only for completeness.

ModelViewWindow.py

This file implements three major classes: `Model`, `View` and `GLUTWindow`. The goal of these classes is to take care of the most common functionalities such as setting up the window, handling user input, setting up view transformations, keeping track of timer for updating frame dependent variables or refresh the display (e.g. when the user performs an action or when an animation is being performed).

`GLUTWindow` is a wrapper around `GLUT`'s windowing functionalities. The `View` class implements the necessary view transformations. The `Model` class implements the base of any model being rendered. The class hierarchy is `GLUTWindow` > `View` > `Model`, where $x > y$ means that x contains and knows about y. The call sequence also follows this pattern. That is, a window can call its child view and a view can call its child which is a model.

How to setup a window

The usual way of setting up the window is to initialize glut, instantiate a model/scene, instantiate the view/camera with a camera specification, and finally call `GLUTWindow` with the view object and some window initialization parameters.

```
from ModelViewWindow import Model, View, GLUTWindow
# ... import other OpenGL related modules ...

# initialize glut
glutInit()
glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA)

# instantiate a scene
# SampleScene is of type Model
scene = SampleScene()

# specify a camera
cam_spec = {'eye' : [0, 1, 5], 'center' : [0, 1, 0], 'up' : [0, 1, 0],
            'fovy': 40, 'aspect': 1.33, 'near' : 1., 'far' : 100.0}

# setup the view using the camera specification
#  and provide the model/scene to be rendered
cam = View(scene, cam_spec)

# create the glut window by providing the
# View object and some initialization parameters
GLUTWindow("Main Camera View", cam,
                window_size = (640, 480), window_pos = (320, 0) )

# glut event loop
glutMainLoop()
```

Initialization sequence and call hierarchy

In python whenever an object is instantiated the `__init__` method gets called. Sometimes we need to initialize OpenGL based objects e.g. `gluQuadrics` which requires calling `gluNewQuadric()`. Initialization of those objects require that OpenGL is initialized (or OpenGL context is initialized) before any of its functions are used. Glut initializes the OpenGL context when `glutCreateWindow` is called inside `GLUTWindow` class. Often times python object instantiation can precede `GLUTWindow` and this may cause an error if an OpenGL based object is instantiated inside `__init__` function. To avoid this we provide an additional `init()` method in `Model`, and `View`. Once `GLUTWindow` is created it calls the `init()` method of its child view which in turn calls the `init()` method of its child model. If a model contains other models then it is necessary to call their `init()` methods from the root model. Naturally if there is no need for initializing OpenGL objects then all model initializations can be done inside the `__init__` method.

The `Model` class implements OpenGL Display List to speed up rendering. Whenever OpenGL receives a command such as `glRotate`, `glTranslate` or `glBegin`, `glEnd`, etc. it has to perform some computations and update its internal data structure. This can become a bottleneck when lots of geometric primitives are rendered in each frame. Display List allows OpenGL to evaluate and store all the rendering commands for rendering a model. Whenever a model needs to be rendered the `Model` classes display function simply calls the Display List.

The `View` class implements some common camera controls. Whenever the user interacts with the window using the keyboard or mouse `GLUTWindow` receives those interactions and calls the relevant function in the `View` class. The control functions are mainly `pan`, `tilt`, `roll`, and corresponding functions for Arcball/Trackball movements.

The `GLUTWindow` class maintains a timer using python's threading mechanism. Whenever the specified timer interval expires it calls all the `timer_func` of the child classes and then updates the window by calling `glutPostWindowRedisplay` with the window id. This triggers the `GLUTWindow.display` function which in turn calls `View.display()`. The `View`'s display function will first perform the appropriate view transformation and then call the child's `display()` function. For external camera's, the `View` class will check if the child is itself a `View` and if so render its camera-frame and then call model's function.