Worksheet 2

Interaction

Exercise 1: Interactors in VTK

a) Animation

This Exercise demonstrates how to setup a simple animation loop. The goal is to let the surface (better said its actor) rotate around its own axis.

1. Open the file `main_surface_ani.py`. Its a modified version of the `main_surface.py` script from the previous lab.

2. Implement a new class called `TimerCallback`.

   (a) The `__init__` method of the class should expect an actor as argument. Then, the actor from the argument is attached to the class (*hint*: `self.actor`). Also, attach one more variable called timer_count to the class that and set its value to zero; `self.timer_count = 0`.

   (b) write a class method `get_angle` that returns an angle based on the `self.timer_count` variable according to

   ```
   1 angle = self.timer_count % 360
   ```

   When our program is run, the `timer_count` variable will be a constantly increasing number. Calling `get_angle` every time-step will return a number that periodically increases from 0 to 359. if you are unfamiliar what the `%` sign does here, read up on the modulo operation.

   This method requires no arguments (beside of course `self`).

   (c) Also write a `execute` method, that expects two arguments, `obj` and `event`. These arguments are mandatory here and will be expected by the internal vtk update routine so we have to provide them even if we dont always use both of them. First, we have to map the current value of `self.timer_count` to the current angle by calling the method from the previous step and storing its return value in a variable called `angle`.

   Now, use the `self.actor` variable of the class (that we attached in 2.a)). More specifically, use its `.SetOrientation()` method to set the orientation to `0, 0, angle`. This means we will change the rotation angle into the z-direction (e.g. a rotation around the z-axis). Read more about the method in VtkUserGuide page Section 4.6.

   Lastly, add the following two lines to the `execute` method:

   ```
   1 obj.GetRenderWindow().Render()
   2 self.timer_count += 1
   ```

The first line here enforces an update of our render window. We have to do this to tell vtk eplicically that something has changed - our rotation angle. the second line increases the `timer_count`.

3. Back in our main pipeline (not the class), after the `# [!] animation` mark, create a new instance of our callable class, add it to our renderwindowinteractor as observer, and also create a repeating timer:

```
1 iren.Initialize() # need to initialize our interactor before we can add
      things
2
3 timer_call = TimerCallback(actor)
4 iren.AddObserver('TimerEvent', timer_call.execute)
5 iren.CreateRepeatingTimer(10)
```

Note: try to take a second here to understand whats going on. On line 3, we create a new instance of the `TimerCallback` class that we just defined. Line 4 is the tricky one, here we add an observer to our interactor, that means we enable a mechanism that calls a specific function whenever a specific even happens. The event that we are observing here is `'TimerEvent'`, but there are other events too that we could react on, for a list of events look e.g. here (link). The function that we want to be executed whenever a `TimerEvent` occurs is the `execute` function from our class. By providing `timer_call.execute` as a second argument without `()` brackets we give a reference to the function itself to the `AddObserver` instead of calling the function. If this python mechanism confuses you, read up on functions as arguments here link.

4. Run your script, fix possible errors and observe the output. How would you change the axis that the plane rotates on? Can you think of other properties of this small scene you could animate? What would you need to change to achieve such a behaviour? Do you understand the purpose of the callables that we used in this and in the previous exercise? Also, note that we provided the function `.execute` to the `.AddObserver` call. Do you understand why we used a class instead of only writing a function here?

b) Cutting Plane (optional, skip for now and finish the other tasks first)

Copy and rename the file `main_volume.py`, that you are familiar with from the first lab. We want to add a plane now and use this plane for cutting through the volume interactively.

1. Add the Plane object, as a global object for now:

```
global plane
plane = vtk.vtkPlane()
```

2. Set the plane as clipping object to the volume mapper

```
<your volume mapper>.AddClippingPlane(plane)
```

3. Add the following function to your script. Its a callback function that gets evaluated automatically when interaction happens. Note that such callback functions are always called in vtk with the syntax `<funcion name>(object, event)`:

```
def my_callback(obj, event):
    global plane
    obj.GetPlane(plane)
```

   This function will be called with an respective object `obj`, and updates the internal status of the plane object.

4. Now we need to create a widget that allows us to interact with the cutting plane. In your code, **after** the initialisation of your renderwindowinteractor, insert the following code snipped:

```
plane_widget = vtk.vtkImplicitPlaneWidget()
plane_widget.DrawPlaneOff()
plane_widget.SetInteractor(iren)
plane_widget.SetInputConnection(reader_src.GetOutputPort())
plane_widget.PlaceWidget()
plane_widget.AddObserver('InteractionEvent' , my_callback)
```

   Note in the last line, we connect our callback function to the plane widget. Also, print the object and event in the `my_callback` function to get a better understanding of the mechanism.

5. Run the script. Don't wonder if you don't see a difference, you probably have to press the letter `i` on your keyboard to start the interaction. Test the interaction and rotate, translate and scale the plane-widget.

6. Making the plane global in order to access it in the callback is not particular pretty. We want to replace the callback function now with a callable class. Therefore, write a python class named `MyCallback` that expects a plane during initialization and attaches it to itself. Then, make the object callable and provide the same functionality in the `__call__` method that our `my_callback` function had. *Note:* callable objects can be written in python like this:

```
class MyCallback:
    def __init__(self, <init arguments>):
        ...

    def execute(self, obj, event):
        ...

# the callable can then be instanced and used like this:
callback_instance = MyCallback(<init arguments>)
```

   lastly, replace the `my_callback` function with `callback_instance.execute` in the `plane_widget.AddObserver()` call.

Exercise 2: Quick introduction of PyQt

So far, we only used vtk directly in the exercises. In order to build a whole application however, it might be feasible to refer to a dedicated GUI-toolkit like e.g. Qt. The usage allows to equip your application with buttons, menus, layout, popups, etc. In this exercise we want to demonstrate how to create a button in qt and also, how to add functionality to it. Open the file `main_qt_inter.py` and implement the following steps.

1. **button**:
   (a) use the `.setText(<text>)` method of the `QPushButton` in order to set the text of the button to 'click me'. A full list of all the methods of the button can be found in the documentation: QtDocumentation. Note that its only available for C++ sadly, so figuring out the python syntax requires a bit of translation.
   (b) Now comes the important part; connect the button to a function that will be called whenever the button is clicked. This can be done in the following way:

```
1 self.button.clicked.connect(<function>)
```

   Concretely, you want to connect the button here to the `.when_clicked` function of the `Application` class. Tip: think about how to pass functions as arguments like in the previous exercise.

2. **layout**: Use the **.addWidget(<QtWidget>)** method of the **QVBoxLayout** class to connect the layout to our button. In other words, our button is a `QtWidget`.

3. **frame**: Use the **.setLayout(<layout>)** method of the **QFrame** class to connect the frame to our layout. Also, note that we set our frame as the central widget of the `Application`.

4. **when_clicked** whenever this function is called, increase the `self.counter` variable by 1. Also, display the value of the counter on the button by calling the `.setText` function from the button again.

5. **more buttons**: Add at least one more button that has an own counter and displays its own click-count.

When you run this script, you should see a Window that contains one/many button(s) with the text 'click me'. When you click one button, it should display a number that tells you how often the button has been clicked.

Exercise 3: Controlling VTK properties with Qt Interface

1. install **Qt Designer** (not Qt creator[1])
   (a) **Linux:** link stackoverflow
   (b) **Windows/Mac:** Link to page with downloads (hint: not tested, use at your own risk)
   (c) **PyCharm** (not required, can run as standalone): link
2. run the designer (e.g. by typing `designer` in the cmd)
3. create a new project of the type `Main Window`
4. familiarize yourself a bit with the Qt designer tool. Its useful for creating your own interfaces. Introductions can be found here.
5. Note: you can always preview your application `Edit` → `Preview`
6. Rebuild the layout that is displayed in the screenshot (Figure 1). Make sure that you rename the widgets properly (see the figure in the Object inspector).
7. make your layout resizable by right clicking into the window without having anything selected and then select `Lay out` → `Lay out Vertically`. Confirm that everything resizes correctly in the preview.
8. Save the projects `.ui` file in our lab working directory: `File` → `Save` as `vtk_layout.ui`
9. Inspect the `vtk_layout.ui` file with a text editor and understand its content.
10. Open the `main_qt_vtk.py` and load the corrent `vtk_layout.ui` file in `uic.loadUi`
11. run the file and confirm that everything opens and resizes correctly (no functionality of the widgets just now).
12. Import vtk `import vtk` and also do `from vtk.qt.QVTKRenderWindowInteractor import QVTKRenderWindowInteractor`. In addition `from random import random`
13. insert the lines from the following code snipped at 13)

```
1 self.vtk_widget = QVTKRenderWindowInteractor(self.vtk_frame)
2 self.vtk_layout.addWidget(self.vtk_widget)
3
4 self.iren = self.vtk_widget.GetRenderWindow().GetInteractor()
5 self.ren = vtk.vtkRenderer()
6 self.vtk_widget.GetRenderWindow().AddRenderer(self.ren)
```

[1] Qt creator can be used also but we're not discussing it in this course and I can also not help you with that.

Worksheet 2

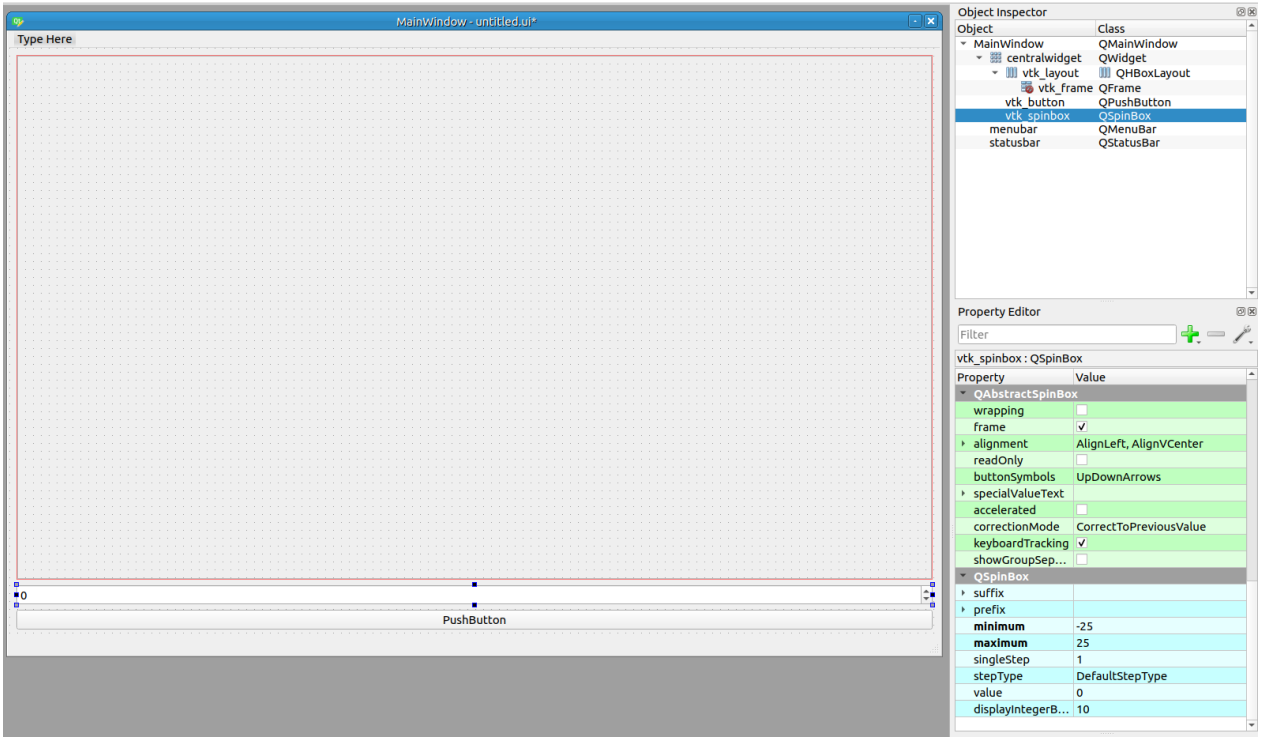

Figure 1: Screenshot of the Layout

14. also, add the two already known commands that start the interactor after `self.show()`

```
self.iren.Initialize()
self.iren.Start()
```

15. run it again and make sure that you see a dark vtk area that resizes correctly.

16. Run and inspect the minimal vtk pipeline in `main_mini.py`. Then, integrate it into out Qt Application that we worked on so far (`main_qt_vtk.py`). Note, pay attention that you insert everything in a place where it belongs and don't add elements that are already there. Concretely, we have already implemented a Renderer, interactor and renderwindow, so we don't need to have more than one.

17. Attach the actor to the `Ui` class (make it a member) by changing all references from `actor` to `self.actor`.

18. Add the following two member functions to the `Ui` class:

```
def change_color(self):
    r, g, b = random(), random(), random()
    prop = self.actor.GetProperty()
    prop.SetColor(r, g, b)
    self.vtk_widget.Render()

def change_position(self):
    self.actor.SetPosition(self.vtk_spinbox.value(), 0, 0)
    self.vtk_widget.Render()
```

19. Whenever our button is clicked, we want to change the color. Therefore, connect the button (hint, its available as self.vtk_button) in the same way we did it in the previous exercise. Note here that this name corresponds to the name that was set in qt designer.

20. Lasly, connect also the spinbox: to the `change_position` method:
`self.vtk_spinbox.valueChanged.connect(self.change_position)`

21. Run the application and make sure that the interactions work correctly (the button switches the color and the spinbox controls the horizontal position)

Here, we just scratched the surface of possible interactions. There are much more Ui-elements available in Qt, for reference see link1 and link2. Also, there are of course much more vtk properties that can be adjusted, e.g. how would you design an interface to load files into your application? By now, you should be able to connect the functionality from Qt and VTK and figure out more concrete use-cases on your own.

Exercise 4: Enabling stereo rendering in vtk

This is a very short exercise. It only demonstrates how stereo rendering can be enabled (you will need that for the final presentation on the stereo-projector).

1. Copy, rename, and run `main_surface.py`. You know this script already from the first lab.

2. Add the following four lines to the bottom of the script, before `ren_win.Render()`:

```
ren_win.GetStereoCapableWindow()
ren_win.StereoCapableWindowOn()
ren_win.SetStereoRender(1)
ren_win.SetStereoTypeToAnaglyph()
```

3. Run again and see if you can see any change. *Note:* You can find all the different stereo-modes that vtk supports here: link. For the final presentation, the mode needs to be set to `SetStereoTypeToCristalEyes`.