SciKit-SurgeryTutorial01 Documentation

Stephen Thompson

Nov 09, 2022

Contents

Author(s): Stephen Thompson. Contributor(s): Miguel Xochicale, Mian Asbat Ahmad and Matt Clarkson.

This is the SciKit-Surgery Augmented Reality Tutorial. It will teach you how to write an augmented reality application, showing a rendered model on top of a live video feed.

The SciKit-Surgery Augmented Reality Tutorial is part of the SciKit-Surgery software project, developed at the Wellcome EPSRC Centre for Interventional and Surgical Sciences, part of University College London (UCL). SciKit-Surgery is a suite of Python modules developed to accelerate the translation of basic science into clinical applications.

The SciKit-Surgery Augmented Reality Tutorial has been tested with Python 2.7 and Python 3.6 and Python 3.7.

CHAPTER 1

Purpose

This tutorial guides the user in producing a simple augmented reality application using some of the SciKit-Surgery libraries. The tutorial is intended to show how SciKit-Surgery can be used to quickly develop applications that include;

- data acquisition,
- data processing,
- data display,
- how SciKit-Surgery enables the use of 3rd party Python libraries.

To remain accessible to novice programmers the tutorial tries to avoid issues such as parsing command line arguments, event loops and GUI design.

1.1 Cloning

You can clone the repository using the following command:

```
git clone https://github.com/SciKit-Surgery/SciKit-SurgeryTutorial01
```

1.2 Contributing

Please see the contributing guidelines.

1.3 Useful links

- Source code repository
- Documentation

CHAPTER 2

Licensing and copyright

CHAPTER 3

Acknowledgements

Supported by Wellcome and EPSRC.

CHAPTER 4

Contents

4.1 Introduction

This is the SciKit-Surgery Augmented Reality Tutorial. SciKit-Surgery aims to support users in developing software applications for surgery. The aim of this tutorial is to introduce the user to SciKit-Surgery. After completing the tutorial the user will be able to;

- make a small augmented reality that shows a rendered surface model overlaid on a live video,
- write an algorithm to move the rendered model,
- write an algorithm to track an ArUco tag in the live video and "attach" the rendered model to the feature.

The tutorial makes use of the SciKit-Surgery library SciKit-SurgeryUtils to create a simple overlay window, showing a VTK model over a video stream. The last part of the tutorial uses SciKit-SurgeryArUcoTracker and SciKit-SurgeryCore to use the motion of a tracked marker to move the model. The tutorial has been tested with Python 3.6 and 3.7 on Linux, Windows, and Mac. and Python 2.7 on Linux.

4.1.1 Augmented Reality in Surgery

Augmented reality is the process of overlaying virtual models onto live video. Augmented reality has useful applications in surgery, where unseen anatomy can be overlaid on the surgical scene, to help the surgeon find anatomy of interest. The example below is from the SmartLiver system, developed using the NifTK platform.

Making an augmented reality application from scratch can be quite complicated. The developer will require an application framework to handle display, threading and user interface, something to provide video streaming, and finally a model renderer. The SciKit-Surgery package scikit-surgeryutils simplifies the process by integrating QT (PySide2), OpenCV, and VTK into a simple to use Python library. This tutorial will guide the user in creating an augmented reality application in around 70 lines of code.

4.1.2 Installation

Step 1: You'll need SciKit-SurgeryUtils installed on your system. Provided you have Python installed on your system you should be able to run . . .

```
pip install scikit-surgeryutils
```

to install SciKit-SurgeryUtils and its dependencies (including SciKit-SurgeryCore). For the third part of the tutorial you'll also need SciKit-SurgeryArUcoTracker

```
pip install scikit-surgeryarucotracker
```

If you don't have Python installed, we recommend downloading an installer for your platform directly from python.org.

4.1.3 Virtual environments

Virtualenv, venv, conda or pyenv can be used to create virtual environments to manage python packages. You can use conda env by installing conda for your OS (conda_installation) and use the following yml file with all dependencies.

```
## Create scikit-surgerytutorial01VE.yml in your favorite location with the following␣
↪content:
##
##  scikit-surgerytutorial01VE.yml
##
## Some useful commands to manage your conda env:
## LIST CONDA ENVS: conda list -n *VE # show list of installed packages
## UPDATE CONDA: conda update -n base -c defaults conda
## INSTALL CONDA EV: conda env create -f *VE.yml
## UPDATE CONDA ENV: conda env update --file *VE.yml --prune
## ACTIVATE CONDA ENV: conda activate *VE
## REMOVE CONDA ENV: conda remove -n *VE --all

name: scikit-surgerytutorial01VE
channels:
  - defaults
  - conda-forge #vtk; tox;
  - anaconda #coverage; scipy;
dependencies:
  - python=3.7
  - numpy>=1.17.4
  - vtk=8.1.2
  - tox>=3.26.0
  - pytest>=7.1.2
  - pylint>=2.14.5
  - pip>=22.2.2
  - pip:
     - PySide2>=5.14.2.3
     - scikit-surgerycore>=0.1.7
     - scikit-surgeryutils>=1.2.0
     - scikit-surgeryarucotracker>=0.1.1
     - opencv-python-headless
```

Step 2: You should now be able to follow the tutorial, using the code snippets contained herein.

4.2 Making a simple model overlay application

In the first instance, let's just get a 3D rendering of a model, overlaid on live video from your webcam, something like this . . .

4.2.1 00 - Simple overlay application

Using your favourite text editor or Python development environment (e.g., **'pycharm'_**, **'vscode'_**, etc), create a new file called vtkoverlay_app.py or similar under a new directory applications or your preferred name.

Start with some import statements

```
import sys
from PySide2.QtWidgets import QApplication
from sksurgeryutils.common_overlay_apps import OverlayBaseWidget
```

scikit-surgery provides an OverlayBaseWidget module that creates a qtwidget showing a live stream from a video source, overlaid with a rendered surface model. scikit-surgery leaves the update method unimplemented so that the user can implement their own version in an child class.

```
#create an OverlayApp class, that inherits from OverlayBaseWidget
class OverlayApp(OverlayBaseWidget):
```

and implement a minimal update method

```
def update_view(self):

    #read a new image from the video source
    _, image = self.video_source.read()

    #copy the image to the overlay window
    self.vtk_overlay_window.set_video_image(image)

    #and render
    self.vtk_overlay_window.Render()
```

Now we build the application itself.

You'll need a surface model (stl, vtk, vtp), which you can put in a directory named "models". You can download the model used in the video above from the project repository, or use a model of your own.

```
if __name__ == '__main__':
  #first we create an application
  app = QApplication([])

  #then an instance of OverlayApp. The video source
  #is set when we create the instance. This is an index
  #starting at 0. If you have more than one webcam, you can
  #try using different numbered sources
  video_source = 0
  viewer = OverlayApp(video_source)

  #Set a model directory containing the models you wish
  #to render and optionally a colours.txt defining the
```

(continues on next page)

(continued from previous page)

```
    #colours to render in.
    model_dir = '../models'
    viewer.add_vtk_models_from_dir(model_dir)

    #start the viewer
    viewer.show()
    viewer.start()

    #start the application
    sys.exit(app.exec_())
```

Now run the application with

```
python vtkoverlay_app.py
```

or similar. If successful you should see a live video stream overlaid with a rendered surface model, something like the video at the top of the page. Congratulations. If not you can download a finished example and compare. Play around with it, see what happens if you delete some line or change part of the update method.

Next we will add some code to the update loop to move the rendered model for each frame update.

4.3 Adding some model motion to overlay

For image guided interventions we typically need some control over the position of model elements. Here we add a few lines of code to our update function to make the overlaid model move. You should end up with something like this . . .

4.3.1 01 - Add some movement to the models

Create a copy of vtkoverlay_app.py and call it vtkoverlay_with_movement_app.py or similar.

Edit the update method for the OverlayApp class, to call a new method called _move_model.

```
def update_view(self):
     _, image = self.video_source.read()

     #add a method to move the rendered models
     self._move_model()

     self.vtk_overlay_window.set_video_image(image)
     self.vtk_overlay_window.Render()
```

Then add a new method called _move_model to the class.

```
def _move_model(self):
     #Iterate through the rendered models
     for actor in self.vtk_overlay_window.get_foreground_renderer().GetActors():
         #get the current orientation
         orientation = actor.GetOrientation()
         #increase the rotation around the z-axis by 1.0 degrees
         orientation = [orientation[0] , orientation[1], orientation[2] + 1.0]
         #add update the model's orientation
         actor.SetOrientation(orientation)
```

Leave the rest of the file as is, and try running the application with

```
python vtkoverlay_with_movement_app.py
```

or similar. If successful you should see a live video stream overlaid with a rendered surface model. The surface model should slowly rotate, like in the video at the top of the page. Congratulations.

Note that you can still use the VTK interactor to move the camera around or change the model representation. Have a play around and see how it interacts with the model rotation.

You can download a finished example and compare. Play around with it, experiment with different ways to move the model or to change the opacity etc.

Next we will add some code to detect an ArUco marker and "pin" the model to it

4.4 Detecting a feature to control model motion

So far we haven't performed any data processing, which is a key element of any surgical AR system. Typically we might get tracking information from an external tracking system, for example using SciKit-SurgeryNDITracker. For this demo, as you're unlikely to have an NDI tracker, we'll use SciKit-SurgeryArUcoTracker which uses computer vision to track a tag.

We should end up with a 3D rendering that follows a tag as you move it in front of the camera. Something like . . .

4.4.1 02 - Add a feature detector and follower

You'll need an ArUco tag to track, print it out in A4 this one .

Create a copy of vtkoverlay_with_movement_app.py and call it vtk_aruco_app.py or similar.

Add an import statement for SciKit-SurgeryArUcoTracker,

```
from sksurgeryarucotracker.arucotracker import ArUcoTracker
```

And an import statement for SciKit-Surgery's transform manager.

```
from sksurgerycore.transforms.transform_manager import TransformManager
```

We'll also need NumPy to handle arrays;

```
import numpy
```

Set up SciKit-SurgeryArUcoTracker in the __init__ function of OverlayApp Class.

```
def __init__(self, image_source):
      """override the default constructor to set up sksurgeryarucotracker"""

      #we'll use SciKit-SurgeryArUcoTracker to estimate the pose of the
      #visible ArUco tag relative to the camera. We use a dictionary to
      #configure SciKit-SurgeryArUcoTracker

      ar_config = {
          "tracker type": "aruco",
          #Set to none, to share video source with OverlayBaseApp
```

(continues on next page)

(continued from previous page)

```
            "video source": 'none',
            "debug": False,
            #the aruco tag dictionary to use. DICT_4X4_50 will work with
            #../tags/aruco_4by4_0.pdf
            "aruco dictionary" : 'DICT_4X4_50',
            "marker size": 50, # in mm
            #We need a calibrated camera. For now let's just use a
            #a hard coded estimate. Maybe you could improve on this.
            "camera projection": numpy.array([[560.0, 0.0, 320.0],
                                              [0.0, 560.0, 240.0],
                                              [0.0, 0.0, 1.0]],
                                             dtype=numpy.float32),
            "camera distortion": numpy.zeros((1, 4), numpy.float32)
            }
        self.tracker = ArUcoTracker(ar_config)
        self.tracker.start_tracking()

        #and call the constructor for the base class
        if sys.version_info > (3, 0):
            super().__init__(image_source)
        else:
            #super doesn't work the same in py2.7
            OverlayBaseApp.__init__(self, image_source)
```

Edit the update method for the OverlayApp class, to call a new method called _aruco_detect_and_follow. Replace the call to method self._move_model() with self._aruco_detect_and_follow().

```
def update_view(self):
     _, image = self.video_source.read()

     #add a method to move the rendered models
     self._aruco_detect_and_follow(image)

     self.vtk_overlay_window.set_video_image(image)
     self.vtk_overlay_window._RenderWindow.Render()
```

Then add a new method called _aruco_detect_and_follow to the class.

```
def _aruco_detect_and_follow(self, image):
        """Detect any aruco tags present using sksurgeryarucotracker
        """

        #tracker.get_frame(image) returns 5 lists of tracking data.
        #we'll only use the tracking matrices (tag2camera)
        _port_handles, _timestamps, _frame_numbers, tag2camera, \
                        _tracking_quality = self.tracker.get_frame(image)

        #If no tags are detected tag2camera will be an empty list, which
        #Python interprets as False
        if tag2camera:
            #pass the first entry in tag2camera. If you have more than one tag
            #visible, you may need to do something cleverer here.
            self._move_camera(tag2camera[0])
```

Delete the _move_model method and replace it with a new _move_camera method

```
def _move_camera(self, tag2camera):
    """Internal method to move the rendered models in
    some interesting way"""

    #SciKit-SurgeryCore has a useful TransformManager that makes
    #chaining together and inverting transforms more intuitive.
    #We'll just use it to invert a matrix here.
    transform_manager = TransformManager()
    transform_manager.add("tag2camera", tag2camera)
    camera2tag = transform_manager.get("camera2tag")

    #Let's move the camera, rather than the model this time.
    self.vtk_overlay_window.set_camera_pose(camera2tag)
```

Leave the rest of the file as is, and try running the application with

```
python vtk_aruco_app.py
```

or similar. If successful you should see a live video stream overlaid with a rendered surface model, similar to the video at the top of the page. When you hold the printed ArUco tag in front of the camera, the model should approximately follow it.

You may notice that the model appears and disappears at certain distances from the camera. This is because we haven't updated the renderer's clipping planes to match the new model position. This can be easily fixed by adding the following code to the update method

```
self.vtk_overlay_window.set_camera_state({"ClippingRange": [10, 800]})
```

Maybe you can do something more sophisticated.

Lastly you will notice that the model doesn't precisely follow the tag. This may be because we haven't calibrated the camera, we just took a guess, so the pose estimation will be wrong. Also we have not set the camera parameters for the VTK renderer, so this will not match the video view.

You can download a finished example of this tutorial file.

You can also download the completed tutorial, either using git;

```
git clone https://github.com/SciKit-Surgery/scikit-surgerytutorial01
```

or by downloading the files directly from

https://github.com/SciKit-Surgery/scikit-surgerytutorial01

That completes this tutorial. Please get in touch with any feedback or issues. You can use the issue tracker at the Project homepage.