Using and Visualizing Projective Cameras in VTK

Release 0.00

David Doria

February 17, 2010

Rensselaer Polytechnic Institute, Troy NY

Abstract

This document presents a set of classes (vtkPhysicalCamera, vtkImageCamera) to enable a camera and an image acquired by that camera to be visualized in a 3D scene. Two situations where one would want to visualize camaras with associated images are range data analysis and 3D scene reconstruction from images (structure from motion). The classes presented in this paper are implemented using tools from VTK.

Contents

1 Introduction

Two increasingly popular technologies require dealing with 3D data and associated images simultaneously. LiDAR scanners produce a 3D point cloud of a scene, while at the same time taking several images of the scene. 3D reconstruction algorithms such as Structure From Motion (SFM) and Multi View Stereo (MVS) take a set of images as input and produce 3D point clouds. In both cases, it is useful to be able to visualize the images relative to the 3D scene. We propose a set of classes as an addition to VTK which allow the user to visualize a camera and an image associated with that camera in 3D space.

2 A Data Structure for Storing, Using, and Visualizing Camera Calibration Parameters - vtkPhysicalCamera

vtkPhysicalCamera is a class for storing intrinsic (focal length, image center) and extrinic (rotation matrix, translation vector) camera parameters. It provides a function, *ProjectPoint* to project a 3D point on to an image (forward projection). It also provides a function, *GetRay*, to compute the 3D ray from the camera center through a pixel given in image coordinates (backward projection).

Looking at the image from the camera center (which is looking down the $+z$ axis), the coordinate system is shown in Figure 1 where w is the image width, h is the image height, and coordinate pairs are specified as (x, y).

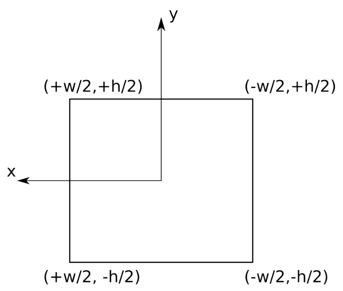

Figure 1: Coordinate system of camera and image.

3 Demonstrations

3.1 Forward Projection

Consider the following camera:

Rotation: 45 degrees around the y axis (rotates the z axis toward the x axis):

$$R = \begin{pmatrix} cosd(45) & 0 & sind(45) \\ 0 & 1 & 0 \\ -sind(45) & 0 & cosd(45) \end{pmatrix} = \begin{pmatrix} .707 & 0 & .707 \\ 0 & 1 & 0 \\ -.707 & 0 & .707 \end{pmatrix}$$

Translation: $C = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$

Focal length: $f = 200$

Principal point: $\begin{pmatrix} 0 \\ 0 \end{pmatrix}$

The intrinsic parameter matrix is therefore:

$$K = \begin{pmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 200 & 0 & 0 \\ 0 & 200 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

From the simple projective camera model, we know:

$$P = KR[I| - C] = [KR| - KRC] = \begin{pmatrix} 141.42 & 0 & 141.42 & -565.6854 \\ 0 & 200 & 0 & -400 \\ -.707 & 0 & .707 & -1.4142 \end{pmatrix}$$

Consider a point

$$X = \begin{pmatrix} 10.0 \\ 20.0 \\ 30.0 \end{pmatrix}$$

The projection of this point by the camera is:

$$x = PX = \begin{pmatrix} 5091.169 \\ 3600 \\ 12.728 \end{pmatrix}$$

Dividing by the third component,

$$x = PX = \begin{pmatrix} 400 \\ 282.84 \end{pmatrix}$$

Octave code

The octave code to reproduce the above calculations is below:

```
R=[cosd(45) 0 sind(45); 0 1 0; -sind(45) 0 cosd(45)]
C=[1;2;3]
f=200
K=[200 0 0; 0 200 0; 0 0 1]
P=[K*R -K*R*C]
P=[K*R -K*R*C]
X=[10;20;30;1]
p=P*X
p=p/p(3)
```

Code Snippet

The same operation (projecting the point $(1,2,3)$ with the camera defined above) is computed using vtk-PhysicalCamera.

```
#include <vtkSmartPointer.h>
#include <vtkMatrix3x3.h>
#include <vtkPointSource.h>
#include <vtkMath.h>

#include "vtkPhysicalCamera.h"

void CreateRotationY(double degree, vtkSmartPointer<vtkMatrix3x3> R);

int main (int argc, char *argv[])
{
  //create a point to project
  vtkSmartPointer<vtkPointSource> pointSource =
      vtkSmartPointer<vtkPointSource >::New();
  pointSource->SetNumberOfPoints(1);
  pointSource->SetCenter(1.0, 2.0, 3.0);
  pointSource->SetRadius(0);
  pointSource->Update();

  vtkPolyData* polydata = pointSource->GetOutput();

  // Setup camera parameters
  // Create a rotation matrix
  vtkSmartPointer<vtkMatrix3x3> r =
      vtkSmartPointer<vtkMatrix3x3 >::New();
  CreateRotationY(45, r);

  double cameraCenter[3] = {10.0, 20.0, 30.0};

  // Create a camera
  vtkSmartPointer<vtkPhysicalCamera> camera =
      vtkSmartPointer<vtkPhysicalCamera >::New();
  camera->SetR(r);
  camera->SetCameraCenter(cameraCenter);
  camera->SetFocalLength(200.0);

  //projection
  double p[3];
  polydata->GetPoint(0,p);
  double pixel[2];
  camera->ProjectPoint(p, pixel);
  cout << "The projection of (" << cameraCenter[0] << ", " << cameraCenter[1] <<
  return EXIT_SUCCESS;
}

void CreateRotationY(double degree, vtkSmartPointer<vtkMatrix3x3> R)
{
  //Rx rotates the y-axis towards the z-axis
  //Ry rotates the z-axis towards the x-axis
  //Rz rotates the x-axis towards the y-axis
  R->Identity();
  double ang = vtkMath::RadiansFromDegrees(degree);
  R->SetElement(0,0, cos(ang));
  R->SetElement(0,2, sin(ang));
  R->SetElement(2,0, -sin(ang));
  R->SetElement(2,2, cos(ang));

}
```

3.2 Backward Projection

If $P = [M|P4]$, then the ray through a pixel $x = \begin{pmatrix} x \\ y \end{pmatrix}$ is

$$C + \alpha M^{-1}\begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

where α is a positive constant (to ensure the ray is pointing "toward the scene").

To demonstrate this, we use the inverse of the forward projection demonstration described above. We wish to find the ray through pixel $\begin{pmatrix} 400 \\ 282.84 \end{pmatrix}$.

M, the first 3x3 block of P, is

$$M = KR = \begin{pmatrix} 141.42 & 0 & 141.42 \\ 0 & 200 & 0 \\ -.707 & 0 & .707 \end{pmatrix}$$

The ray through the pixel is then:

$$\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} + \alpha M^{-1}\begin{pmatrix} 400 \\ 282.84 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} + \alpha \begin{pmatrix} .707 \\ 1.414 \\ 2.121 \end{pmatrix}$$

Normalizing the direction to unit magnitude, we have

$$\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} + \alpha \begin{pmatrix} 0.26726 \\ 0.53452 \\ 0.80178 \end{pmatrix}$$

Octave code

The octave code to reproduce this calculation is simply

```
M=P(1:3, 1:3)
inv(M)*p %the direction of the ray
```

Code Snippet

The same operation (projecting the point $(1,2,3)$ with the camera defined above) is computed using vtk-PhysicalCamera.

```
#include <vtkSmartPointer.h>
#include <vtkMatrix3x3.h>
#include <vtkRenderWindow.h>
#include <vtkRenderer.h>
#include <vtkPolyData.h>
#include <vtkPolyDataMapper.h>
#include <vtkXMLPolyDataReader.h>
#include <vtkAxesActor.h>
#include <vtkImageData.h>
#include <vtkOrientationMarkerWidget.h>
#include <vtkRenderWindowInteractor.h>
#include <vtkInteractorStyleTrackballCamera.h>
#include <vtkJPEGWriter.h>
#include <vtkCellArray.h>
#include <vtkMath.h>
#include <vtkPointSource.h>
#include <vtkCameraActor.h>
#include <vtkCamera.h>

#include "vtkPhysicalCamera.h"

void CreateRotationY(double degree, vtkSmartPointer<vtkMatrix3x3> R);

int main (int argc, char *argv[])
{
  //Setup camera parameters

  //Create a rotation matrix
  vtkSmartPointer<vtkMatrix3x3> r =
      vtkSmartPointer<vtkMatrix3x3 >::New();
  CreateRotationY(45, r);

  double cameraCenter[3] = {0.0, 0.0, 0.0};

  // Create a camera
  vtkSmartPointer<vtkPhysicalCamera> camera =
      vtkSmartPointer<vtkPhysicalCamera >::New();
  camera->SetR(r);
  camera->SetCameraCenter(cameraCenter);
  camera->SetFocalLength(200.0);

  double pix[2] = {400.0, 282.84};
  double ray[3];
  camera->GetRay(pix, ray);

  cout << "The ray through pixel (" << pix[0] << "," << pix[1] << " is (" << ray

  return EXIT_SUCCESS;
}

void CreateRotationY(double degree, vtkSmartPointer<vtkMatrix3x3> R)
{
  //Rx rotates the y-axis towards the z-axis
  //Ry rotates the z-axis towards the x-axis
  //Rz rotates the x-axis towards the y-axis
  R->Identity();
  double ang = vtkMath::RadiansFromDegrees(degree);
  R->SetElement(0,0, cos(ang));
  R->SetElement(0,2, sin(ang));
  R->SetElement(2,0, -sin(ang));
  R->SetElement(2,2, cos(ang));

}
```

3.3 Visualizing Camera Position and Orientation

Figure 2 shows the camera (Large X,Y,Z) relative to the world coordinate frame (X_o,Y_o,Z_o). (The axes in the lower left corner is an orientation widget and is there only to help when the world coordinate axes are not drawn).

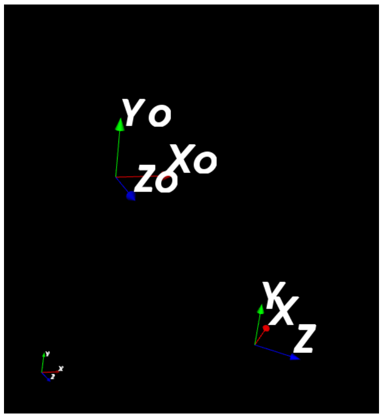

Figure 2: Camera location and orientation relative to the world coordinate system.

To enable the camera to be displayed in the scene, you must simply call:

```
camera->UpdateCameraActor();
renderer->AddActor(camera->GetAxesActor());
```

4 Displaying a 2D Image in a Meaningful 3D Position - vtkImageCamera

vtkImageCamera is derived from vtkPhysicalCamera. It serves as a camera with an associated image. The image can be drawn on any slice of the camera frustum, specified by the *DistanceFromCamera* member variable. A quad is created by casting rays through the corner pixels and finding the coordinates of the points *DistanceFromCamera* from the camera center along each ray. The image stored in the class is then texture mapped onto this quad. We provide an example with a slider to control the slice of the view frustum that is displayed (how far the image is from the camera). A screenshot of this demonstration is shown in Figure 3. We use a capital 'R' as the image because it is completely asymmetric, so orientation correctness is easy to spot.

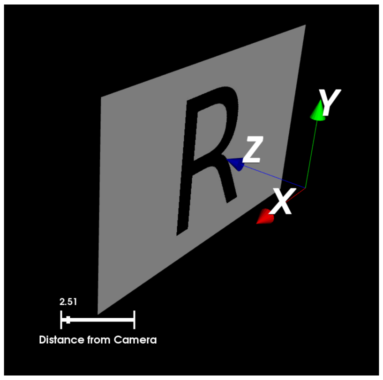

Figure 3: An example camera and its associated image.

4.1 Code Snippet

Since vtkImageCamera derives from vtkPhysicalCamera, most of the functionality (setting the camera parameters, etc) is identical. The main additional feature is setting the image. This is demonstrated below:

```
vtkSmartPointer<vtkJPEGReader> reader =
        vtkSmartPointer<vtkJPEGReader>::New();
reader->SetFileName(filename.c_str());
reader->Update();

camera->SetTextureImage(reader->GetOutput());

vtkSmartPointer<vtkRenderer> renderer =
        vtkSmartPointer<vtkRenderer>::New();
camera->SetRenderer(renderer);
camera->UpdateCameraActor();
camera->UpdateImageActor();
```

4.2 Back-projected Rays

We provide an interactive demonstration of computing and visualizing the ray through an image pixel. A screenshot is shown in Figure 4. It was necessary to use Qt for this example as we needed one renderer to use a 2D interactor style and the other renderer to use a 3D interactor style.

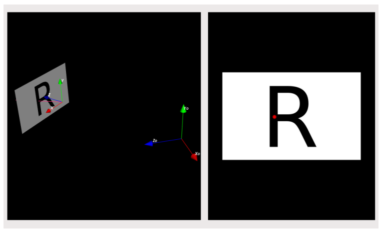

Figure 4: A screenshot of the interactive backprojected rays demonstration.

The image is displayed in the right renderer. When the user moves the red dot (a vtkSphereWidget), the ray through the selected pixel is displayed in the left renderer.

5 Conclusion and Future Work

We have presented a set of classes for visualizing cameras with associated images. In the future we will include a derived camera class to remove radial and tangential lens distortion if the distortion coefficients are provided.