Paraview_script_python

August 26, 2020

1 Script python 3.X pour ParaView

1.1 Etape 1, importer le module de ParaView, qui est contenu dans l'installation de ParaView

Cette étape est faite automatiquement au démarrage de ParaView, mais vous avez à l'ajouter lorsque vous créez votre propre script. Le module définit une fonction pour chaque source, *reader*, filtre et *writer*. Les fonctions ont le même nom que ce qui est affiché dans l'interface graphique, donc ça permet de mieux s'y retrouver.

Chaque fonction crée un objet dans le pipeline, qui va apparaître dans l'explorateur de pipeline et retourne un objet (python) qui est un proxy qui peut être utilisé pour interroger et manipuler les propriétés d'un objet dans le pipeline.

Le défi est de se souvenir de la différence entre un objet python et un objet visuel dans ParaView (ou objet de pipeline). Les objets Python et ParaView peuvent avoir le même nom ou pas et un objet présent dans Python peut ne pas être affiché dans l'explorateur de Paraview.

ATTENTION! Si vous suivez un guide en ligne qui est écrit en python 2.X, vous allez voir des différences dans la syntaxe!

```
In [ ]: from paraview.simple import *
```

```
In [ ]: # pour avoir une liste des fonctions de Paraview, il faut faire
        # dir(paraview.simple)
        # ou
        # help(paraview.simple)
```

1.2 Etape 2, faire afficher une sphère

On peut faire apparaître une sphère, comme on l'a déjà fait en cliquant sources-> sphere, mais cette fois-ci en le tapant explicitement.

```
In [ ]: sphere = Sphere() #la fonction Sphere() cree
        # l'objet sphere dans le pipeline
        Show() # la fonction Show() indique que les objets
        # doivent etre visibles, son oppose est Hide()
        # laisser le contenu des parentheses vide.
        Render() # la fonction Render() force Paraview a
        # faire le rendu pour afficher quelque chose dans
        # la fenetre de visualisation
```

```
ResetCamera() # a faire lorsqu'on a un script python.
# Paraview s'ajuste automatiquement à la presence
# d'objets lorsqu'ils sont ajoutes avec l'interface graphique,
# mais pas lorsqu'un script python est
# execute
valeur = 3 # cet objet est valide pour python, mais
# n'apparait pas dans le pipeline
```

1.3 Etape 3, creer un filtre et le faire afficher

On veut maintenant ajouter un filtre. Ça se fait de manière pratiquement identique à la création d'objet. Pour Python, ils sont aussi des objets et si on se souvient, ils sont de même classe que les sources et readers dans ParaView. Par défaut, le dernier objet créé dans le pipeline va servir d'intrant pour la fonction du filtre.

```
In [ ]: Hide() # On fait disparaitre ce qu'on a fait de notre rendu
        shrink = Shrink() # la fonction Shrink() va faire rapetisser
        # les polygones de la sphere et lui donner
        # une allure explosee
        Show()
        Render()
```

1.4 Etape 4, ajuster les paramètres des objets

Nous avons vu dans l'interface qu'il est possible de modifier les paramètres des objets dans le panneau de proˆpriétés. Il est également possible de modifier ces paramètres dans le code, comme par exemple pour le cas de notre sphère qui est anguleuse plutôt que lisse.

Dans python, il est possible de connaitre les attributs d'un objet en executant dir()

```
In [ ]: dir(sphere)
```

Demandons a python de nous donner la résolution Theta de l'objet sphere. Le point dans "sphere.ThetaResolution" sert a spécifier qu'on veut l'attribut ThetaResolution et rien d'autre. La réponse devrait etre 8, qui est le paramètre par défaut.

```
In [ ]: print(sphere.ThetaResolution)
```

Nous allons maintenant changer cette résolution explicitement en spécifiant l'attribut et la valeur que nous désirons lui donner.

```
In [ ]: sphere.ThetaResolution = 16 # on rajoute des polygones
        Render() # le nouveau rendu a un air different
```

Nous allons maintenant reprendre la fonction pour le filtre Shrink, qui permet de réduire les polygones de la sphère. En inspectant avec dir(), on voit qu'il n'y a qu'une seule propriété au filtre Shrink: le Shrink Factor. C'est cette propriété que nous allons influencer en changeant la valeur de l'attribut ShrinkFactor de 0.5 à 0.25.

```
In [ ]: dir(shrink)
        print(shrink.ShrinkFactor) # la valeur par defaut est 0.5
        shrink.ShrinkFactor = 0.25 # on change le ShrinkFactor
        # de notre objet/filtre shrink
        Render() # on voir des tous petits triangles
```

1.5 Etape 5, créer des *branching pipelines*

Nous désirons maintenant voir ce qui arrive lorsque nous creéns un pipeline qui ne va pas de A-B-C mais A-B1-C1 A-B2-C2 A-B3-C3. Nous avons un objet sphere et un objet shrink, qui représentent un objet dans le pipeline et un filtre dans le pipeline dans ParaView. Nous allons maintenant ajouter un filtre qui va extraire la structure en cage a poule d'un objet visé.

Nous utilisons la fonction ExtractEdges, qui représente le filtre Extract Edges et nous spécifions dans les parenthèses que nous voulons avoir la sphère en entrée en mettant Input=sphere comme argument. Il serait possible de specifier l'objet d'entrée plus tard, mais il est mieux de le mettre dès le début. Tous les filtres ne peut pas accepter n'importe quel objet en entrée et si on laisse un code rouler sans supervision en oubliant d'avoir spécifié l'objet d'entrée pour une de ces filtres en particulier, le code pourrait s'arrêter avec une erreur si l'objet d'entrée ne fait pas partie de ce que le filtre peut accepter.

```
In [ ]: wireframe = ExtractEdges(Input=sphere)
        Show()
        Render()
```

Le résultat de la source sphère a maintenant le filtre shrink et le filtre wireframe attachés à lui. C'est un exemple de ce qui est décrit comme un pipeline *fan out*. Certains filtres peuvent avoir plusieurs filtres en intrant, selon un pipeline appellé *fan in*. Par exemple, on peut utiliser les filtres shrink et wireframe en intrant pour GroupDatasets, qui mettra les deux dans un seul groupe.

```
In [ ]: group = GroupDatasets(Input=[shrink, wireframe])
        Show()
```

On peut donc maintenant cacher les filtres qui ne nous intéressent plus, puisque que nous avons un filtre qui combine les deux.

```
In [ ]: Hide(shrink)
        Hide(wireframe)
        Render()
```

1.6 Étape 6, faire des modifications de paramàtres directement sur un objet

Nous avons modifié des paramàtres sur les filtres, mais nous pouvons faire la même chose sur les objets. Il suffit d'utiliser les arguments à l'intérieur des parenthèses lors de la création de l'objet. Par exemple, nous pouvons créer une sphère qui a la bonne résolution dès son premier affichage.

```
In [ ]: sphere = Sphere(ThetaResolution=360, PhiResolution=180)
        Show()
```

1.7 Étape 7, identifier les objets actifs

Les objets actifs sont ceux sur lesquels nous aurions cliqué dans l'explorateur de pipeline si nous avions utilisé l'interface graphique. Quand on utilise l'interface graphique en même temps que le terminal python, ils partagent le même objet actif. Dans les étapes précédentes, c'est l'objet actif qui était choisi par défaut lorsqu'aucun intrant n'était spécifié. Lorsqu'on créait un nouvel objet, cet objet devenait l'objet actif.

```
In [ ]: GetActiveSource() # pour connaitre la source actuellement active
        #SetActiveSource() # pour changer la source active
        GetActiveView() # pour connaitre la source actuellement active
        #SetActiveView() # pour changer la source active
```

2 Lire et sauvegarder des fichiers

2.1 Etape 1, Lecture de fichiers

L'équivalent d'ouvrir un fichier dans l'interface graphique est de créer un *reader* dans le script Python. Comme les sources et les filtres, il faut créer un objet dans python qui va devenir un proxy pour l'objet dans le pipeline de ParaView.

Nous allons pouvoir faire un essai avec le fichier ExodusII du disque chauffant en rotation.

```
In [ ]: reader = OpenDataFile('Examples/disk_out_ref.ex2')
        Show()
        Render()
        ResetCamera()
```

2.2 Étape 2, Interrogation des attributs de champs

Nous connaissons déjà un certain nombre des propriété de l'objet créé par le fichier disk_out_ref.ex2, mais avec Python, il y a en plus certaines propriétés et méthodes communes à tous les objets Python qui servent de proxy pour les objets dans le pipeline ParaView, comme par exemple *PointData* et *CellData*.

Ces deux propriétés agissent comme des *dictionnaires* dans le sens pythonesque du terme. On les utiliser avec les méthodes (encore un terme Python) *GetNumberOfComponents*, qui retourne la taille de chaque valeur de champs, *GetName*, qui retourne le nom du champs et *GetRange* qui retourne les valeurs minimales et maximales.

Par exemple, utilisons PointData à partir de notre reader pour disk_out_ref.ex2:

```
In [ ]: pd = reader.PointData
```

On a maintenant créé un objet *pd* qui contient reader.PointData. Nous allons pouvoir aller chercher des informations comme par exemple le contenu des champs *Pres* et *V* avec lesquels nous avons travaillé abondamment dans l'interface graphique.

```
In [ ]: print(pd.keys())
        print(pd['Pres'].GetNumberOfComponents())
        print(pd['Pres'].GetRange())
        print(pd['V'].GetNumberOfComponents())
```

Nous pourrions même passer à travers chacune des composantes de pd pour extraire les informations qui nous intéressent à l'aide d'une boucle for.

```
In [ ]: for ai in pd.values():
            print(ai.GetName(), ai.GetNumberOfComponents(),)
            for i in xrange(ai.GetNumberOfComponents()):
                print(ai.GetRange(i),)
            print
```

2.3 Étape 3, les représentations

Les représentations sont la colle entre l'objet dans le pipeline et le visuel qui nous est présenté. Dans l'interface graphique, les représentations sont créées automatiquement et nous pouvons les changer d'un simple clic de souris. Avec un script python, c'est la commande Show() qui crée un proxy de la représentation. Comme les filtres et les sources, on peut mettre Show() dans un objet et même sauvegarder son résultat. C'est aussi pour ça que chaque vue dans l'interface graphique peut avoir un rendu de représentation différent de la voisine.

Par exemple, si on voulait changer la couleur des données, considérant que nous n'avons pas sauvegardé Show() dans un objet, il faudrait d'abord aller chercher la représentation actuelle:

```
In [ ]: readerRep = GetRepresentation()
```

Ensuite, nous pourrions aller modifier les éléments qui nous intéressent, comme la propriété *specular* qui rend l'objet très luisant.

```
In [ ]: readerRep.DiffuseColor = [0, 0, 1]
        readerRep.SpecularColor = [1, 1, 1]
        readerRep.SpecularPower = 128
        readerRep.Specular = 1
        Render()
```

Il serait aussi possible d'utiliser la fonction ColorBy() pour spécifier les champs selon lesquels on veut ajuster la couleur. Nous connaisson ColorBy() pour l'avoir utilisé dans l'interface graphique. Il faut par la suite utiliser UpdateScalarBars() pour ajuster la barre de couleur dans l'annotation.

```
In [ ]: ColorBy(readerRep, ('POINTS', 'Pres'))
        UpdateScalarBars()
        Render()
```

2.4 Étape 4, les vues

Dans l'interface graphique, les vues sont placées pour nous automatiquement. Avec Python, il faut les spécifier plus explicitement. C'est Render() qui retourne une vue. On peut vouloir changer les couleurs de fond des vues, ou bien organiser les vues de differentes façon, comme on l'a fait avec l'interface graphique. On peut aussi contrôler les angles de caméra, les annotations, etc. s'il est impossible d'y avoir accès par l'interface graphique.

Par exemple, pour changer la couleur de fond, il faut commencer par sauvegarder la vue active (puisque nous n'avons pas utilisé de fonction comme CreateView('nom'), CreateRenderView ou CreateXYPlotView).

```
In [ ]: view = GetActiveView()
```

Nous allons changer la couleur de fond en allant chercher et modifier 'Background' dans view:

```
In [ ]: view.Background = [0, 0, 0]
        view.Background2 = [0, 0, 0.6]
        view.UseGradientBackground = True
        Render()
```

On peut finalement prendre la position de la caméra et utiliser une petite boucle pour créer une animation.

```
In [ ]: x,y,z = view.CameraPosition
        print(x,y,z)
        for iter in xrange(0,10):
            x = x + 1
            y = y + 1
            z = z + 1
            view.CameraPosition = [x,y,z]
            print(x,y,z)
            Render()
```

2.5 Étape 5, sauvegarder les résultats

Plusieurs options s'offrent à vous pour sauvegarder les éléments que vous désirez:

Pour les données d'un filtre, il faut commencer par CreateWriter() et ensuite mettre à jour le proxy pour le writer avec la méthode UpdatePipeline(). Ça équivaut à cliquer File -> Save Data.

```
In [ ]: plot = PlotOverLine() # graphique sur une ligne

        # Definition du point 1 et 2 de la ligne
        plot.Source.Point1 = [0,0,0]
        plot.Source.Point2 = [0,0,10]

        # Writer cree
        writer = CreateWriter('Examples/plot.csv')

        # Writer mis a jour
        writer.UpdatePipeline()
```

Pour sauvegarder une image, on peut exécuter SaveScreenshot() et spécifier un chemin et une extension.

```
In [ ]: # On cree une vue
        plotView = CreateView('XYChartView')
        # On met "plot" dedans
        Show(plot)
        Render()

        # On sauvegarde une image de la vue active
        SaveScreenshot('Examples/plot.png') # ex: un png
```

Nous n'allons pas sauvegarder d'animation, mais vous pourriez le faire avec:

```
In [ ]: WriteAnimation('chemin/vers/animation.extension')
```