Visualização tridimensional com o VTK

Manipulação 3D no Python

www.sxc.hu – wolf friedmann

Atualmente, o mundo dos objetos 3D já faz parte do cotidiano: é só reparar no uso que a indústria do entretenimento faz dessa tecnologia. Aprenda como usar o Python para lidar com 3D, com o VTK.

POR ANA M. FERREIRO E JOSÉ A. GARCÍA RODRÍGUEZ

A representação gráfica em 3D oferece a possibilidade de se criar mundos virtuais. Com a visualização 3D, torna-se possível explorar e entender sistemas complicados rapidamente, graças à evolução das linguagens orientadas a objetos, que permitem criar programas de melhor qualidade e de fácil manutenção.

Entre as diferentes ferramentas de visualização, representação e processamento de imagens 3D, vale destacar o VTK (*Visualization Toolkit*) [1], uma biblioteca de código aberto (implementada em *C++*) com *wrappers* para *TCL*, *Python* e *Java*, permitindo o desenvolvimento de aplicações completas, de um modo eficiente e com o uso de scripts simples. Por tudo isso, o VTK é hoje bem utilizado na visualização 3D nas áreas médica, industrial, na reconstrução de superfícies a partir de digitalização a laser ou nuvens de pontos desorganizados etc.

A seguir, veremos os conceitos básicos usados pelo VTK para criarmos uma cena e, através de uma série de exemplos desenvolvidos em Python, vamos criar nossos próprios cenários de visualização.

Instalação

Para nossos exemplos, é preciso ter instalado o Python e o VTK (com

Quadro 1: Como compilar o VTK

Para instalar o VTK a partir do código fonte, que podemos baixar no site do VTK [3], podemos escolher uma das versões em *.tgz* ou acessar o *CVS* e fazer o download da última versão. No segundo caso, criaremos uma pasta chamada "VTK" em `/opt` e, para ter acesso ao repositório CVS, digitamos:

```
cvs -d :pserver:anonymous@public. ➥
kitware.com:/cvsroot/VTK login
(responder com a senha: vtk)
```

Para baixar o código fonte, teclamos:

```
cvs -d :pserver:anonymous@public.kitware.com:/cvsroot/VTK checkout VTK
```

Em qualquer um dos casos, para compilar o VTK precisamos do *Cmake*, que pode ser obtido em www.cmake.org/HTML/Download.html. Se preferir, também pode-se obter o Cmake pelo CVS:

```
cvs -d :pserver:anonymous@www.cmake.org:/cvsroot/CMake ➥
login password: cmake
```

Para baixá-lo, digitamos:

```
cvs -d :pserver:anonymous@www.cmake.org:/cvsroot/CMakeco CMake
```

Uma vez que tenhamos baixado e instalado o Cmake, podemos compilar o VTK. Para isso, devemos entrar na pasta onde está o VTK e teclar a partir da linha de comando: `cmake -i`.

Um prompt vai perguntar o que se quer compilar. Devemos prestar atenção e, quando aparecer a pergunta sobre a opção de se instalar os wrappers para Python, responder afirmativamente (já que a resposta não é automática). Então, basta teclar `make` e depois `make install`.

suporte para Python). Além disso, a placa de vídeo precisa estar com o *OpenGL* funcionando.

Há duas maneiras de instalar o VTK: a primeira é baixar e instalar os pacotes binários. No caso do *Suse*, *Fedora* ou *Mandriva* (*Mandrake*), basta procurar pelos seguintes pacotes RPM (em nosso exemplo, para o Mandrake 10.1):

- vtk-4.2.2-5mdk.i586.rpm
- vtk-python-4.2.2-5mdk.i586.rpm
- vtk-tcl-4.2.2-5mdk.i586.rpm
- vtk-examples-4.2.2-5mdk.i586.rpm
- vtk-devel-4.2.2-5mdk.i586.rpm

Esses pacotes podem ser baixados através do repositório *RedIris* [2]. A outra maneira é compilar o código fonte, conforme explicado no **quadro 1**: "Como compilar o VTK".

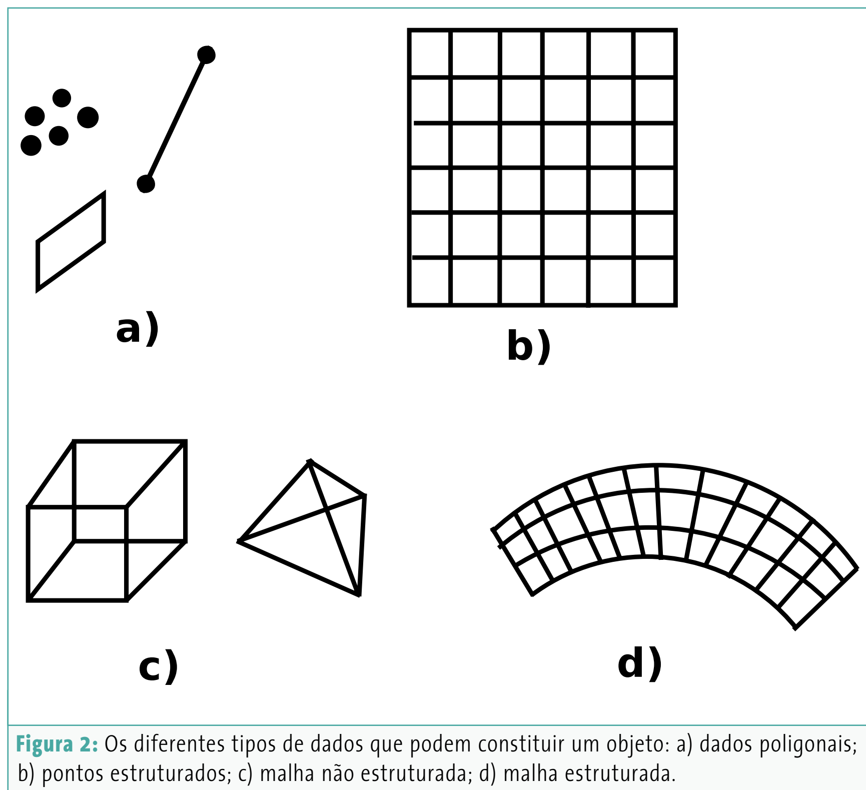

Figura 2: Os diferentes tipos de dados que podem constituir um objeto: a) dados poligonais; b) pontos estruturados; c) malha não estruturada; d) malha estruturada.

Modelos de objetos VTK

Para os iniciantes no mundo da visualização 3D, vamos resumir uma explicação sobre a estrutura do VTK, pois isso permite compreender melhor cada passo deste tutorial.

Imagine uma cena de um desenho animado como, por exemplo, "A Era do Gelo". Se nos concentrarmos em uma única seqüência, veremos personagens animados, luzes de diferentes tonalidades, câmeras que mudam o ponto de vista, características dos personagens (cor, forma etc)… Mesmo que você não acredite, todos esses conceitos são a base da visualização gráfica. Vejamos como é essa estrutura.

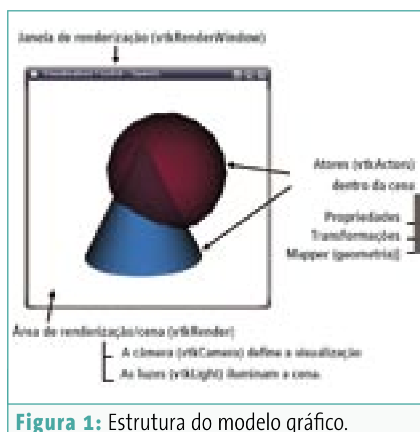

Figura 1: Estrutura do modelo gráfico.

A biblioteca VTK foi projetada a partir de dois modelos, claramente distintos: o modelo gráfico e o modelo de visualização.

- *Modelo gráfico* – Captura as principais características de um sistema gráfico 3D, de modo fácil de compreender e utilizar (**figura 1**). A abstração está baseada na indústria do cinema. Os modelos básicos que constituem esse modelo são: `vtkRenderer`, `vtkRenderWindow`, `vtkLight`, `vtkCamera`, `vtkProp`, `vtkProperty`, `vtkMapper`, `vtkTransform` (a **tabela 1** descreve cada um desses objetos).
- *Modelo de visualização* – O papel do modelo gráfico é o de transformar dados gráficos em imagens, enquanto que o modelo de visualização transforma informações em dados gráficos. Isso significa que o modelo de visualização é o responsável por construir a representação geométrica que é renderizada através do modelo gráfico. O VTK se baseia na aproximação dos dados para transformar a informação em dados gráficos. Há dois tipos básicos de objetos envolvidos na aproximação (descritos na **tabela 2**): `vtkDataObject` e `vtkProcessObject`.

Os diferentes tipos de dados que podem constituir um objeto são: pontos, retas, polígonos, pontos estruturados, malhas estruturadas, malhas não estruturadas etc (**figura 2**).

Primeira cena

Já estamos prontos para construir a nossa primeira cena. Vamos nos colocar no papel do diretor de cinema. Nos exemplos a seguir, veremos como empregar as classes que acabamos de descrever. Para isso, assim como dissemos no início, vamos usar instâncias de objetos VTK dentro dos scripts Python. ➟

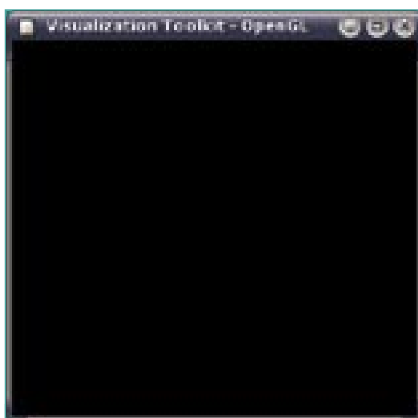

Figura 3: A janela de renderização automática.

Por meio de qualquer editor de texto, criamos um arquivo `cone.py`. A primeira coisa a fazer é importar o pacote VTK do Python. Isso é bastante simples. Basta a seguinte linha: `import vtk`.

Agora já podemos criar instâncias de qualquer objeto VTK, apenas escrevendo `vtk.nome_classe` no código do programa. Precisamos também criar nossa janela de renderização `vtk.vtkRenderWindow`, que chamaremos de `renWin`, e à qual associaremos uma área de renderização `vtk.vtkRenderer` (que denominamos `ren`), através do método `AddRenderer()`. Isso é feito com as linhas de código a seguir:

```
ren=vtk.vtkRenderer()
renWin=vtk.vtkRenderWindow()
renWin.AddRenderer(ren)
iren=vtk.vtkRenderWindowInteractor()
iren.SetRenderWindow(renWin)
```

Para que possamos manipular a câmera por meio do mouse, o objeto `vtkRenderWindowInteractor` (denominado no código como `iren`) deverá ser instanciado. Note que a janela de renderização `renWin` se associa ao objeto de interação `iren` através do método `SetRenderWindow`. Nesse momento, ainda não percebemos a utilidade disso. Paciência, vamos compreender sua importância quando tivermos um "ator" em nosso cenário.

Fechamos o arquivo e, na linha de comando, executamos o programa digitando `python cone.py`. E... nada acontece! Isso ocorre porque devemos iniciar a interação do usuário e indicar que a janela de renderização deve permanecer visível até que o usuário finalize sua execução, fechando-a. Para isso, basta adicionar ao código:

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

Ao executarmos novamente o programa, veremos uma janela negra se abrir, com seus botões de maximizar, minimizar e fechar, e que só será fechada quando o usuário quiser (**figura 3**). Ela será o "armazenador" da nossa pequena cena. Note que as duas linhas de código que acabamos de escrever devem ficar no final do arquivo. As demais linhas que venham a ser escritas deverão ficar logo antes.

Não vamos criar um ator muito complicado, pois queremos ver logo alguma coisa acontecer.

O VTK contém uma série de classes que nos permitem criar objetos tridimensionais simples como esferas (`vtkSphereSource`), cones (`vtkConeSource`) e cilindros (`vtkCilinderSource`), entre muitos outros. Tomaremos como exemplo um cone – no entanto, você pode optar por qualquer um dos outros objetos. O seguinte código cria nosso primeiro "ator":

```
cone=vtk.vtkConeSource()
coneMapper=vtk.vktPolyDataMapper()
coneMapper.SetInput(cone.GetOutput())
coneActor=vtk.vtkActor()
coneActor.SetMapper(coneMapper)
```

Tabela 1: Modelo gráfico

Objeto	Descrição
vtkRenderer	Cria uma área de renderização que coordena luzes, câmeras e atores.
vtkRenderWindow	Classe que representa o objeto dentro do qual se coloca uma ou mais áreas de renderização (vtkRenderer).
vtkLight	Objeto que permite manipular as luzes da cena. Quando se cria uma cena, as luzes são incluídas automaticamente.
vtkCamera	Objeto que controla como uma geometria 3D é projetada dentro da imagem 2D durante o processo de renderização. A câmera tem diferentes métodos que permitem definir o ponto de vista, o foco e a orientação.
vtkProp	Objeto que representa os diferentes elementos (atores) que se situam dentro da cena. Cabem destacar as seguintes sub classes: vtkActor, vtkVolume, vtkActor2D.
vtkProperty	Representa os atributos de renderização de um ator, incluindo cor, iluminação, mapa da estrutura, estilo de desenho e estilo de sombra.
vtkMapper	Representa a definição da geometria de um ator e mapeia os objetos mediante uma tabela de cores (vtkLookupTable). O mapper proporciona a fronteira entre o modelo de visualização e o modelo gráfico.
vtkTransform	Objeto que consiste em uma matriz de transformação 4x4 e métodos para modificar a matriz mencionada. Especifica a posição e orientação de atores, câmeras e luzes.

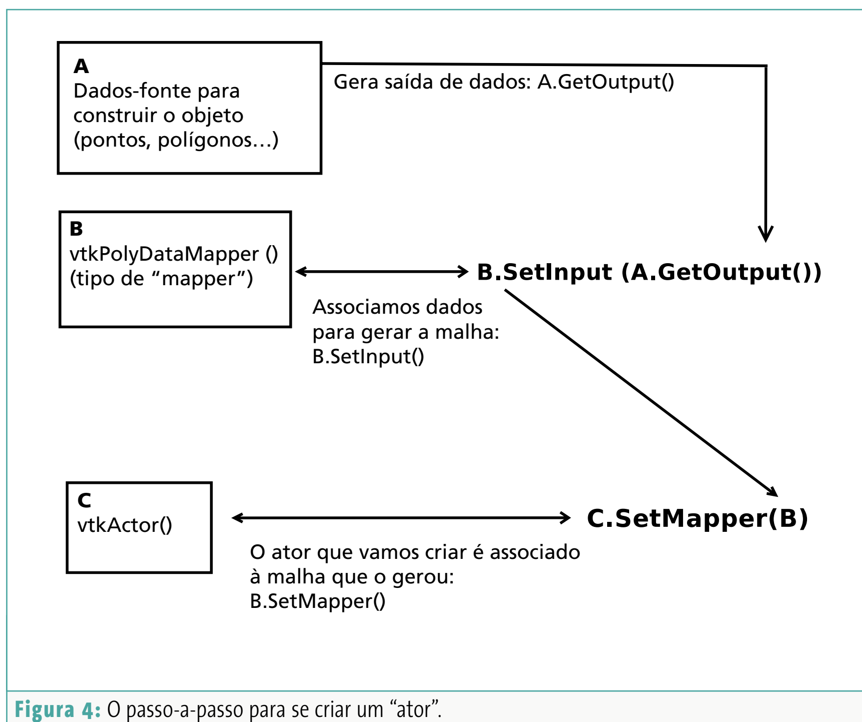

Figura 4: O passo-a-passo para se criar um "ator".

Por meio do objeto `vtk.vtkConeSource`, criamos a representação poligonal de um cone que chamamos de... "cone". A saída do cone (`cone.GetOutput()`) é um conjunto que se associa ao "mapper" (`coneMapper`) (`vtk.vtkPolyDataMapper`) através do método `SetInput()`. Criamos o "ator" (objeto a ser renderizado) ao qual se associa a representação geométrica que resulta em `coneMapper`. Note que os passos indicados aqui são, geralmente, os que precisamos seguir para construir um ator (**figura 4**).

Quando criamos um ator, ele não é inserido automaticamente na cena. É preciso antes adicioná-lo ao Render com um `AddActor` e renderizar a cena posteriormente. Isso é possível com as linhas:

```
ren.AddActor(conoActor)
renWin.Render()
```

Ao executá-lo novamente, iremos visualizar um cone de cor cinza (cor inserida automaticamente) dentro da nossa janela (**figura 5**). Além disso, é nesse momento que a interação com o mouse se mostra importante: com o botão esquerdo é possível rotacionar a câmera; o botão central permite realizar traslados e, com o botão direito, nos aproximamos ou nos afastamos do objeto. Além disso, uma luz foi inserida automaticamente para observarmos os objetos iluminados.

Experimente comentar a linha `renWin. Render()`. O que acontece? Como você já deve ter-se dado conta, o cone não aparece. Isso porque, cada vez que adicionamos um ator, é necessário renderizar a cena, pois, se não a retomamos, é como se não houvéssemos adicionado um novo ator.

Propriedades

Se você seguiu o tutorial até esse ponto já terá criado seu cone cinza. Mas, provavelmente, não está muito satisfeito. Talvez outra cor fosse melhor, como branco e fundo azul, por exemplo. Na seqüência, vamos ver como modificar a janela de renderização, a câmera e as propriedades do ator. No final será possível fazer todas as alterações que desejar.

Você já deve ter observado que a janela de renderização se abre com um tamanho pré-determinado. Para fixar o tamanho da janela é preciso empregar o método `SetSize`, no qual indicamos a altura e largura em pixels: `renWin.SetSize(450,325)`.

Se o que queremos é mudar a cor de fundo da cena (`vtkRenderer`), empregamos o método `SetBackground(RGB)`, no qual informamos a cor desejada

Tabela 2: Modelo de visualização

Objeto	Descrição
vtkDataObject	Classe genérica que permite representar diferentes tipos de dados. Os objetos de dados consistem em estruturas geométricas e topológicas (pontos e células), e também em atributos associados, tais como escalares ou vetores.
vtkProcessObject	Objeto que faz referência aos filtros, que atuam sobre os atores, modificando-os.

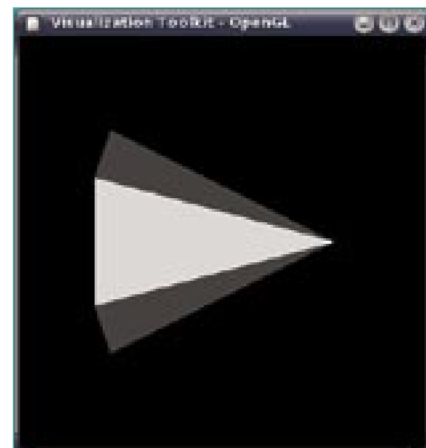

Figura 5: Cone dentro da cena.

no formato RGB. Se quisermos um fundo azul, basta digitar `ren.SetBackground(0.0, 0.0, 1)`.

Conforme já explicado, a área de renderização (`vtkRenderer`) coordena a câmera e as luzes. Através do método `GetActiveCamera()` temos acesso à câmera criada na cena e assim podemos aplicar a ela todos os métodos do objeto `vtkCamera` para modificar a visualização como quisermos. Se o que queremos é que todos os atores sejam vistos em sua totalidade dentro da área de renderização, é preciso chamar o método `ResetCamera()`. Nas linhas seguintes, podemos observar alguns dos métodos relativos à câmera:

```
ren.ResetCamera()
camera=ren.GetActiveCamera()
camera.Azimuth(60)
camera.Pitch(5)
camera.Yaw(5)
camera.Roll(50)
camera.Elevation(20)
camera.Zoom(1.5)
```

Os métodos `Azimuth`, `Pitch`, `Yaw`, `Roll` e `Elevation` são responsáveis por rotacionar a câmera ou o ponto de foco em diferentes posições e tem, como argumento, o ângulo de rotação. O melhor é brincar um pouco com a câmera e ver o que acontece, testando cada um desses métodos em separado.

Por exemplo, para ver qual o efeito do método `Azimuth` aplicado à câmera, comente as linhas de código restantes, caso contrário, estaria misturando métodos de rotação diferentes e, mesmo assim, não seria possível saber direito o que está acontecendo. Não se preocupe se em algum momento o ator sumir de cena. O que está acontecendo é que o ângulo de rotação é tal que é impossível ver o ator dentro da cena. A **figura 6** explica de maneira simplificada como atua cada um desses métodos levando em consideração o foco (representado por uma esfera branca).

A partir desse ponto, comente as linhas de código correspondentes aos métodos que atuam sobre a câmera, deixando só a linha `camera.Zoom(1.5)` ativa. Assim podemos ver uma coisa de cada vez e, logo, poderemos mesclar o código.

Agora sabemos modificar a cena, mas o cone continua sendo visto na cor cinza, um pouco apagado. Para se alterar as propriedades de qualquer ator `vtkActor`, devemos utilizar o método `GetProperty()`, que gera uma instância do objeto `vtkProp`, associada ao referido ator.

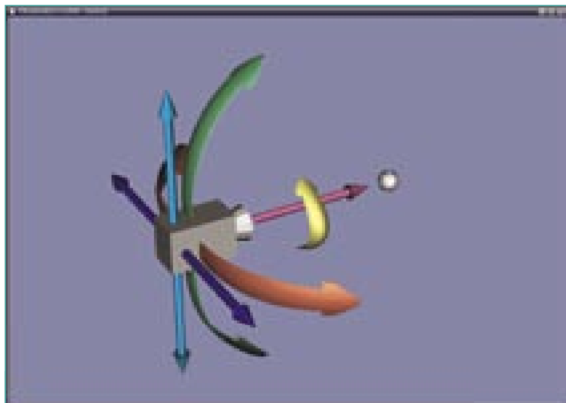

Figura 6: Comportamento dos métodos da câmera a) *Azimuth* - flechas vermelhas; b) *Pitch* - flechas azul-celeste; c) *Yaw* - flechas azul-escuro d) *Elevation* - flechas verdes e) *Roll* - flecha amarela. A esfera branca representa o foco.

As linhas seguintes permitem modificar a cor, transparência e espessura das linhas:

```
conepro=coneActor.➥
GetProperty()
conepro.SetColor(1,0.2,0)
conepro.SetOpacity(0.5)
conepro.SetLineWidth(3)
conepro.SetResolution(40)
conepro.SetRepresentation➥
ToWireframe()
```

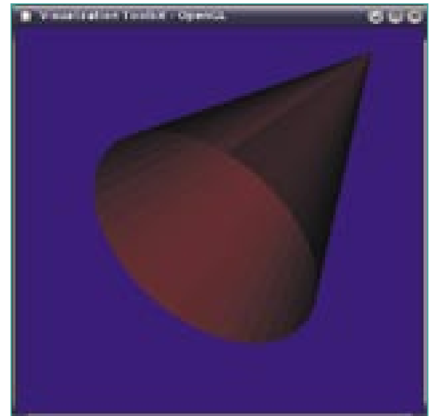

Figura 7: Vista da superfície do cone.

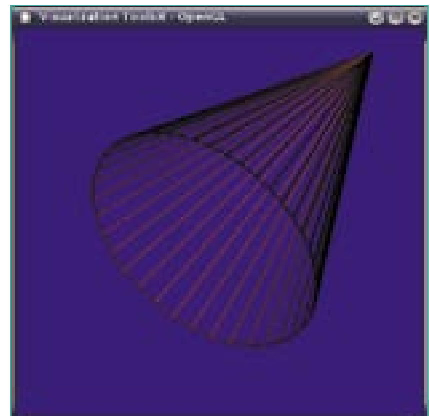

Figura 8: Vista da malha do cone.

A última linha desse código indica que queremos ver a estrutura básica que constitui o ator, ou seja, a malha. O VTK já vem com teclas de atalho associadas à cena: ao pressionar a tecla ***[S]***, todos os objetos são vistos renderizados (**figura 7**). Já ao teclar ***[W]***, vê-se apenas a malha (**figura 8**). Agora é possível observar melhor a diferença entre a malha e a estrutura renderizada. Nada melhor que poder enxergar as coisas!

A linha `conepro.SetResolution(40)` modifica a resolução com a qual se renderiza o cone. Mas esse método não serve para todos os atores, apenas para certos objetos que já são incluídos pelo VTK, tais como esfera (`vtkSphereSource`), cone (`vtkConeSource`), cilindro (`vtkCilinderSource`) etc.

Listagem 1: cone_esfera.py

```
01 import vtk
02
03 # Geramos a estrutura para visualizar um cone
04 cone = vtk.vtkConeSource()
05 coneMapper = vtk.vtkPolyDataMapper()
06 coneMapper.SetInput(cone.GetOutput())
07 coneActor = vtk.vtkActor()
08 coneActor.SetMapper(coneMapper)
09
10 # Criar fonte de esfera, mapeador e ator
11 esfera = vtk.vtkSphereSource()
12 esferaMapper = vtk.vtkPolyDataMapper()
13 esfera.SetPhiResolution(10)
14 esfera.SetThetaResolution(20)
15 esfera.SetCenter(0.3,0.0,0.0)
16 esferaMapper.SetInput(esfera.GetOutput())
17 esferaActor = vtk.vtkActor()
18 esferaActor.SetMapper(esferaMapper)
19 esferaActor.GetProperty().SetColor(0.7,0.0,0.25)
20 esferaActor.GetProperty().SetOpacity(0.75)
21 esferaActor.GetProperty().SetLineWidth(1)
22
23 # Criamos: Renderer, RenderWindow, ➘
  RenderWindowInteractor
24 ren = vtk.vtkRenderer()
25 renWin = vtk.vtkRenderWindow()
26 renWin.AddRenderer(ren)
27 iren = vtk.vtkRenderWindowInteractor()
28 iren.SetRenderWindow(renWin)
29
30 # Adicionamos o ator na área de renderização (Renderer)
31 ren.AddActor(coneActor)
32 ren.AddActor(esferaActor)
33
34 #Fixamos a cor de fundo, o tamanho e damos zoom sobre
35 #a área de Renderização
36 ren.SetBackground(1, 1, 1)
37 renWin.SetSize(450, 425)
38 camera=ren.GetActiveCamera()
39 ##camera.Zoom(1.5)
40
41 coneActor.RotateX(30)
42 coneActor.RotateY(45)
43 conepro=coneActor.GetProperty()
44 conepro.SetColor(0,0.6,1)
45 ##conepro.SetOpacity(0.5)
46 conepro.SetLineWidth(2)
47 ren.ResetCamera()
48 ##camera=ren.GetActiveCamera()
49 camera.Zoom(1.5)
50
51 cone.SetResolution(40)
52
53 iren.Initialize()
54 renWin.Render()
55 iren.Start()
```

Qualquer objeto pode ser rotacionado, escalonado, ter suas dimensões mostradas etc, utilizando-se as propriedades de um `vtkActor` em particular (para mais informações, consulte a ajuda do VTK sobre o `vtkProp3D`, que é a classe pai). Para rotacionar e escalonar nosso cone, usamos as linhas:

```
coneActor.RotateX(30)
coneActor.RotateY(45)
coneActor.SetScale([1,3,2])
```

Agora você já sabe como criar sua própria cena, modificar suas propriedades, adicionar um ator com as opções que quiser e modificar a posição da câmera. No caso de desejar adicionar mais atores à sua janela de renderização, basta seguir o mesmo procedimento empregado na criação do nosso cone. Na **listagem 1** são adicionados à cena um cone e uma esfera que se intersectam (**figura 9**). ■

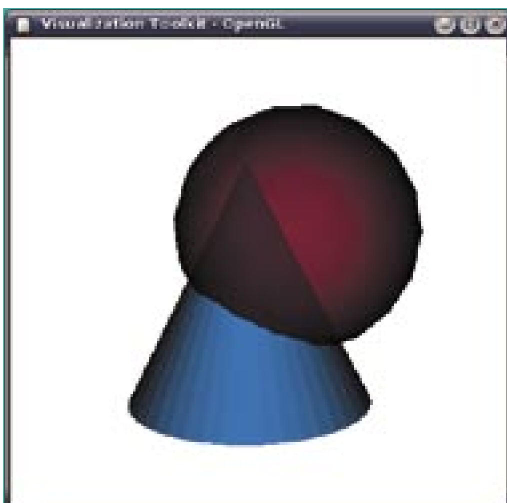

Figura 9: Cena com dois atores em intersecção, uma esfera sobre um cone.

Informações

[1] VTK: public.kitware.com/VTK/

[2] RPMs do VTK no Mandrake: ftp.rediris.es/sites3/carroll.cac.psu.edu/mandrakelinux/official/10.1/i586/media/contrib/

[3] Downloads VTK: www.vtk.org/get-software.php

[4] Enthought. Ferramentas científicas para Python: www.scipy.org

[5] MayaVi: mayavi.sourceforge.net

Autores

Ana M. Ferreiro e José García Rodríguez estudaram matemática, mas são apaixonados por informática e dedicam a ela grande parte de seu tempo.