Visualization Toolkit

1. General description

VTK is an open-source software system for 3D computer graphics, image processing and scientific visualization. Distributed under BSD 3-clause License

2. How to draw any shape?

In general there are two ways of drawing any kind of shape using VTK:

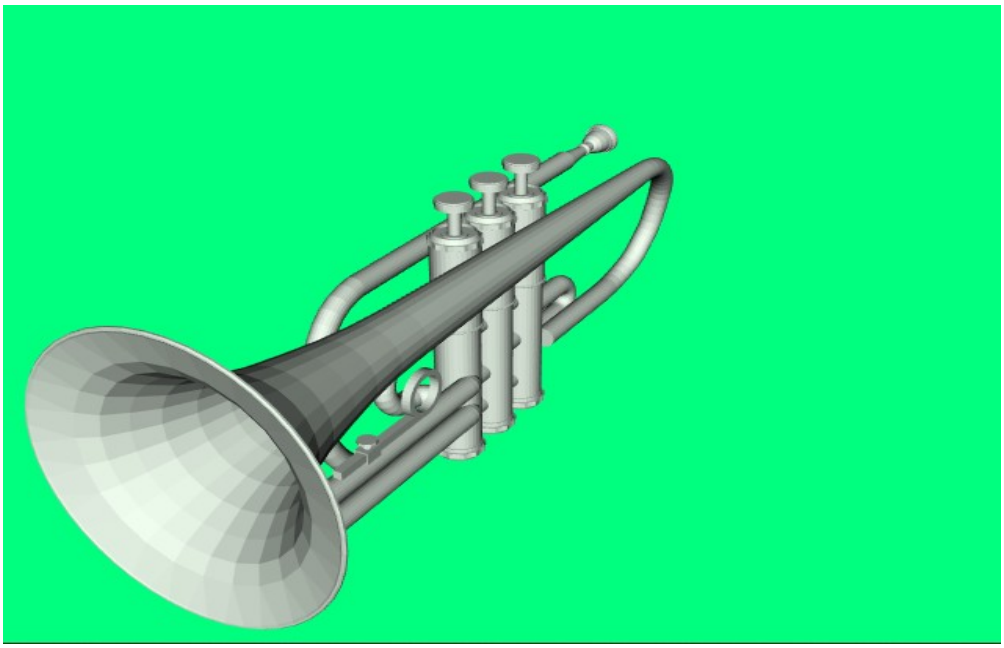
- **Manual drawing** → We can hardcode some polygons but it isn't the most efficiency way of creating any visible objects.
- **Using specified components with supported file format** → VTK has many supported file formats, which help to easily draw wanted shape or image for our visualization.

3. Supported formats:

- **OBJ** → this format contains information about the geometry of 3D object. It can be used with DEM component (Digital Elevation Model), which in general create maps with supported terrain difference and elevation.
  To read file we simply have to create class instance:

```
// Read the file
  vtkNew<vtkDEMReader> reader;
  reader->SetFileName(argv[1]);
  reader→Update();
```

As a result we receive e.g. this kind of image:



- **OBJ with textures** → using VTK it's possible to not only read object shape and geometry, but also its textures.
  To do it we have to use vtkOBJImporter:

```
int main(int argc, char* argv[])
{
 if (argc < 4)
 {
  std::cout
      << "Usage: " << argv[0]
      << " objfile mtlfile texturepath e.g. doorman.obj doorman.mtl doorman"
      << std::endl;
  return EXIT_FAILURE;
 }
 vtkNew<vtkOBJImporter> importer;
 importer->SetFileName(argv[1]);
 importer->SetFileNameMTL(argv[2]);
 importer→SetTexturePath(argv[3]);
```

  As a result we receive object with its textures:

- **Standard input formats** → similar to above-mentioned OBJ case, part of a list of supported standard input formats with examples it's possible to find here: https://kitware.github.io/vtk-examples/site/Cxx/#3d-file-formats

- **Standard importers formats** → similar to above-mentioned OBJ with textures case, part of a list of supported standard importers formats with examples it's possible to find here: https://kitware.github.io/vtk-examples/site/Cxx/#importers

- **Standard output formats** → with VTK it's possible to write data for some types of files, part of a list of supported standard output formats with examples it's possible to find here: https://kitware.github.io/vtk-examples/site/Cxx/#output

- **VTK input formats** → VTK has its own dedicated file formats for input process, part of a list of them with examples it's possible to find here: https://kitware.github.io/vtk-examples/site/Cxx/#input_1

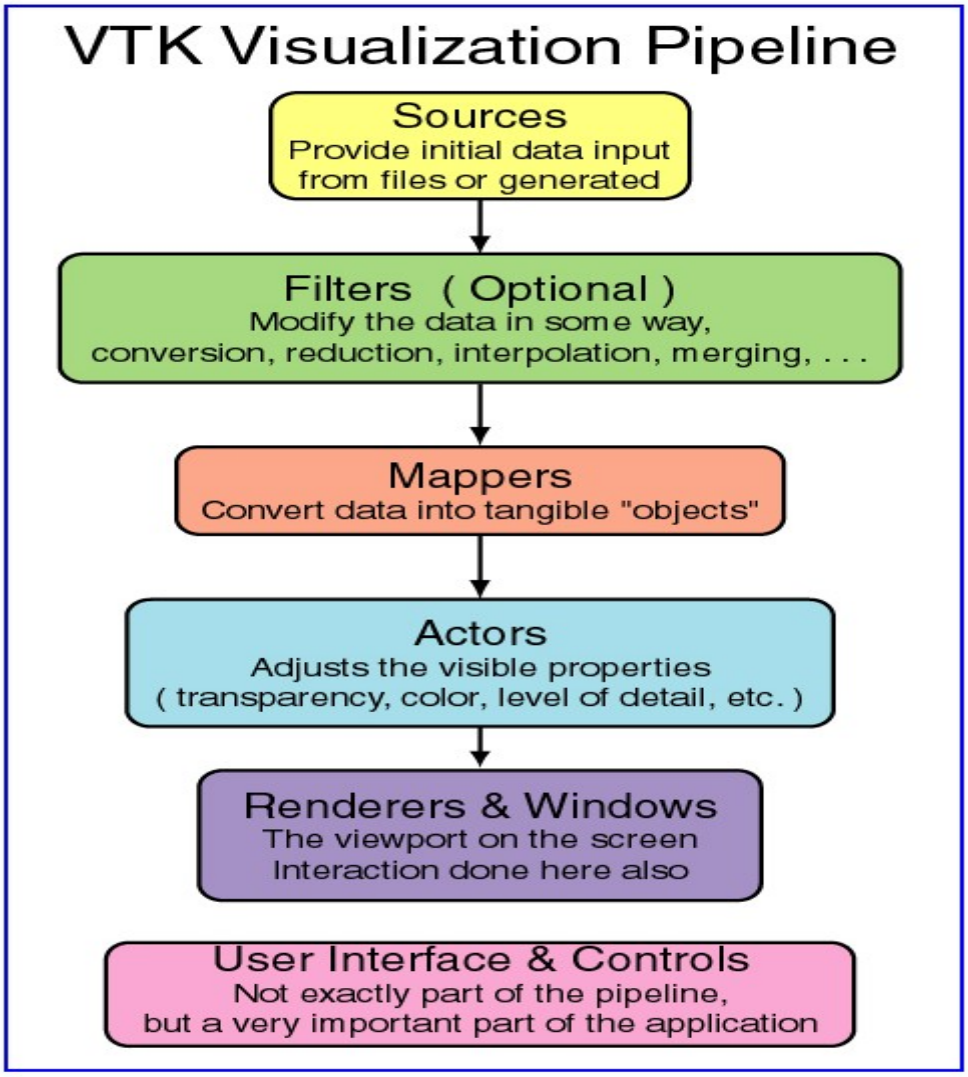
- **VTK output formats** → except of input formats it's also possible to use dedicated output formats, part of the list of them with examples it's possible to find here: https://kitware.github.io/vtk-examples/site/Cxx/#output_1

- **Image input format** → with VTK it's also possible to read image format files, part of the list with examples it's possible to find here: https://kitware.github.io/vtk-examples/site/Cxx/#input_2

- **Image output format** → analogically it's possible to write data for image objects into files, part of a list of supported formats with examples it's possible to find here: https://kitware.github.io/vtk-examples/site/Cxx/#output_2

4. VTK Visualization Pipeline

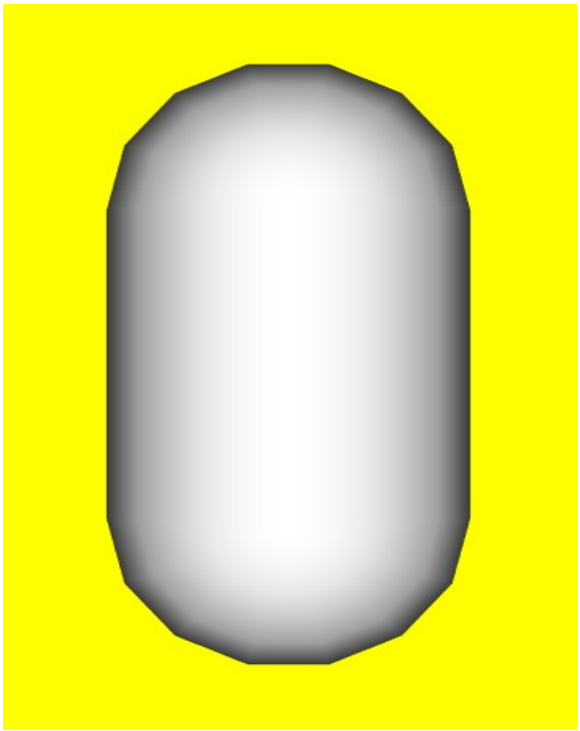
Accurate way to describe VTK image visualization process is to use pipeline metaphor:



Above example come from: https://www.cs.uic.edu/~jbell/CS526/Tutorial/Tutorial.html

According to above-mentioned we can create minimal “HelloWorld” program:

```
int main(int, char*[])
{
/* Source */
 vtkNew<vtkCapsuleSource> cylinder;

/* Mapper */
vtkNew<vtkPolyDataMapper> cylinderMapper;
cylinderMapper->SetInputConnection(cylinder->GetOutputPort());

/* Actor */
vtkNew<vtkActor> cylinderActor;
cylinderActor->SetMapper(cylinderMapper);

/* Renderer and window */
vtkNew<vtkRenderer> renderer;
renderer->AddActor(cylinderActor);
renderer→SetBackground(10, 10, 0);

vtkNew<vtkRenderWindow> renderWindow;
renderWindow->SetSize(300, 300);
renderWindow->AddRenderer(renderer);

vtkNew<vtkRenderWindowInteractor> renderWindowInteractor;
renderWindowInteractor->SetRenderWindow(renderWindow);

renderWindow->Render();
renderWindowInteractor->Start();

return EXIT_SUCCESS;
}
```

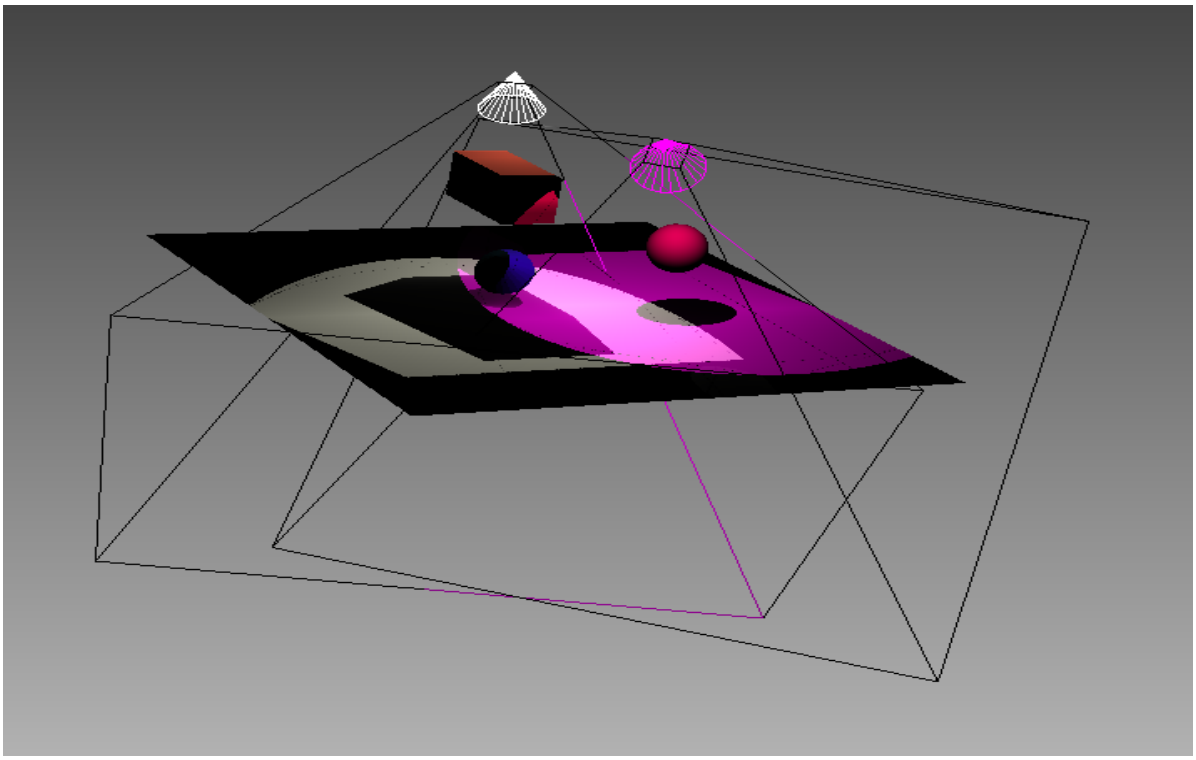


A little bit more complex version of this “HelloWorld” example (e.g. with a description of colors logic in VTK) it’s possible to find here:
https://kitware.github.io/vtk-examples/site/Cxx/GeometricObjects/CylinderExample/

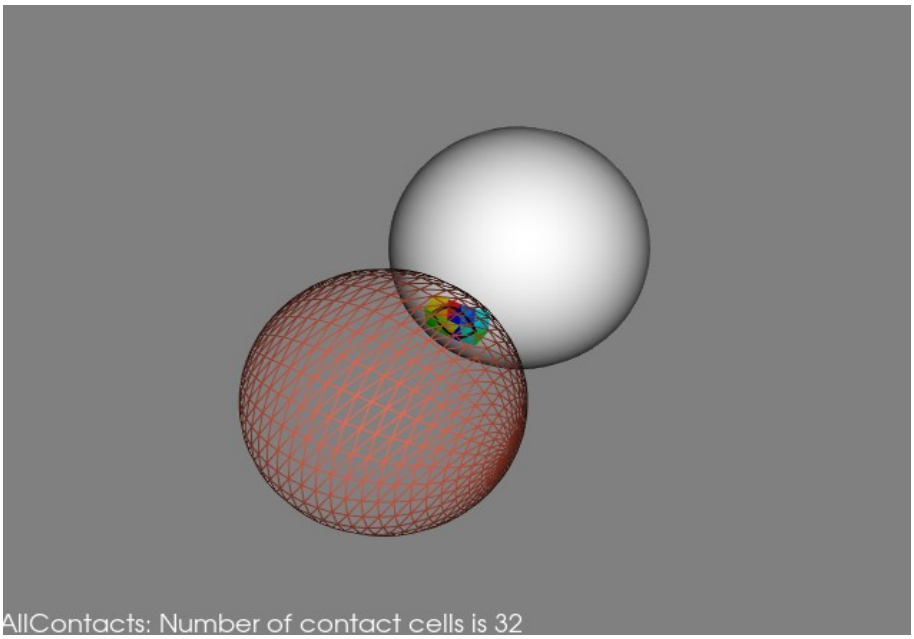
5. What is also possible?

- **Animations** → VTK provides dedicated mechanisms for creating animations, some of examples it's possible to find here: https://kitware.github.io/vtk-examples/site/Cxx/#animation

- **Shadows and Lights** → Both of them are possible to use with VTK.
  Here is example project which use shadows and lights (more projects here: https://kitware.github.io/vtk-examples/site/Cxx/#lighting):



- **Collision** → there's a collision detection mechanism in VTK, which e.g. provide a possibility to detect collision between two actors (example is possible to find here: https://kitware.github.io/vtk-examples/site/Python/Visualization/CollisionDetection/):



DEMO project

Project focused on loading only OBJ objects.

First step was to load .OBJ from files. To do it we use vtkOBJReader (in future we want to change this into vtkOBJImporter which support loading textures) which is a source object that reads Wavefront .obj files. The output of this source object is polygonal data. Because every object needs his own Reader we decided to do by using vector container. Every Reader needs to set file name and then it can load data from files:

```
std::vector<std::string> fileNames{
"../images/desk.obj", "../images/trumpet.obj",
"../images/lamp.obj", "../images/notebook.obj",
"../images/wineglass.obj", "../images/pill.obj",
"../images/pill.obj", "../images/pill.obj",
"../images/pill.obj", "../images/pill.obj",
"../images/pill.obj", "../images/pillbottle.obj",
"../images/floor.obj", "../images/chair.obj"};

std::vector<vtkNew<vtkOBJReader>> vectorReader;

for (const auto &fileName : fileNames) {
	vtkNew<vtkOBJReader> reader;
	reader->SetFileName(fileName.c_str());
	reader->Update();
	vectorReader.emplace_back(std::move(reader));
}
```

To display object on screen first we have to create a mapper which is an abstract class to specify interface between data and graphics primitives. Because it's abstract class we have to subclass this or use some predefined subclass e.g. vtkPolyDataMapper or vtkDataSetMapper. In most of case we were using vtkPolyDataMapper which is a class that maps polygonal data to graphics primitives and serves as a superclass for device-specific poly data mappers, that actually do the mapping to the rendering/graphics hardware/software.

Next step is to connect every mapper Input to reader Output using SetInputConnection() method:

```
std::vector<vtkNew<vtkPolyDataMapper>> vectorMapper;
for (const auto &reader : vectorReader) {
        vtkNew<vtkPolyDataMapper> mapper;
        mapper->SetInputConnection(reader->GetOutputPort());
        vectorMapper.emplace_back(std::move(mapper));
}
```

Then we need to create an actor which is used to represent an entity in a rendering scene. It inherits functions related to the actors position, and orientation from vtkProp. The actor also has scaling and maintains a reference to the defining geometry (e.g. the mapper), rendering properties, and possibly a texture map.

Every actor needs his own mapper. We have to set it by using method SetMapper(vtkMapper *)
This is the method that is used to connect an actor to the end of a visualization pipeline e.g. the mapper.
Then we can transform and move this object. Some object can be placed in the same place.

Next is creating a renderer. Renderer is an object that controls the rendering process for objects. Rendering is the process of converting geometry, a specification for lights, and a camera view into an image. vtkRenderer also performs coordinate transformation between world coordinates, view coordinates (the computer graphics rendering coordinate system), and display coordinates (the actual screen coordinates on the display device). Certain advanced rendering features such as two-sided lighting can also be controlled.

To display window we have to create a rendering window which is a window in a graphical user interface where renderers draw their images. Methods are provided to synchronize the rendering process, set window size, and control double buffering. The window also allows rendering in stereo. The interlaced render stereo type is for output to a VRex stereo projector. All of the odd horizontal lines are from the left eye, and the even lines are from the right eye. The user has to make the render window aligned with the VRex projector, or the eye will be swapped. To this object we have to add Renderer. Which will render whole project.

```
vtkNew<vtkRenderWindow> renderWindow;
renderWindow->AddRenderer(renderer);
renderWindow→SetWindowName("Example");
renderWindow->SetSize(640, 480);
renderWindow->Render();
```

to provide some interaction, we create vtkRenderWindowInteractor which provide platform-independent interaction mechanism for mouse/key/time events. It serves as a base class for platform-dependent implementations that handle routing of mouse/key/timer messages to vtkInteractorObserver and its subclasses. vtkRenderWindowInteractor also provides controls for picking, rendering frame rate, and headlights.

```
vtkNew<CustomMouseInteractorStyle> style;
renderWindowInteractor->SetInteractorStyle(style);

renderWindow->SetSize(640, 480);
renderWindow->Render();

renderWindowInteractor->Start();
```