vtkPainter: An Improved Poly Data Mapper

Kenneth Moreland

Last Updated: November 16, 2004.

1. Purpose of this Document

VTK is often criticized for its slow rendering speed. Although vtkOpenGLPoly-DataMapper has improved significantly, it leaves much to be desired. The vtk-OpenGLPolyDataMapper class passes data to OpenGL with calls to glVertex, glNormal, and the like, which often make speed of the rendering API bound. The vtkOpenGLPolyDataMapper class can also use display lists, but display lists are slow to build and require a significant amount of memory.

The vtkSNL repository contains vtkOpenGLVertexArrayPolyDataMapper. This mapper uses vertex arrays rather than display lists. The vtkOpenGLVertexArray-PolyDataMapper is very efficient for certain types of poly data, such as a collection of only triangles with only point normals and colors. However, slight deviations incur drastic performance hits. Short length triangle strips, polygons of varying size, and cell centered data can all kill the performance of vtkOpenGL-VertexArrayPolyDataMapper. Furthermore, the monolithic code of vtkOpenGL-VertexArrayPolyDataMapper makes it hard to improve the rendering speed for the various conditions that can occur in poly data. Therefore, vtkOpenGLVertex-ArrayPolyDataMapper is a mediocre solution for a general purpose poly data mapper.

This is an informal document to the capture design criteria and ideas for an improved VTK mapper. The code exists and is currently residing in the vtkSNL repository.

2. Mapper Overview

The failure of the design of vtkOpenGLVertexArrayPolyDataMapper was that it picked a rendering method (vertex arrays) that was fast for a certain class of poly data. It then handled all other types of poly data by shoehorning them into rendering methods that did not render them well or using fallback rendering methods that are simply slow to begin with.

To design a better poly data mapper, we should keep two things in mind. First, the layout of the polygons in the poly data is not consistent. Second, there is no rendering method that will be the best for every collection of polygons. The

optimal rendering method can change depending on whether the polygons are only triangles, arranged in triangle strips, or a zoo of various polygon types. Furthermore, the optimal rendering method can change depending on whether properties are defined on points, cells, or both. To make the problem even more complicated, rendering performance can change from system to system, meaning we may never know the best rendering method *a-priori*.

To ensure the flexibility of the new poly data mapper, we use the *Strategy* design pattern from the gang-of-four book.[1] The pattern would make the poly data mapper work as follows. The poly data mapper object simply maintains a *context*, which is state of the rendering process. The poly data mapper object delegates the algorithm it uses to render to a separate *strategy* object. The *strategy* object is defined by an interface in an abstract class with concrete implementations inheriting from it. The algorithm used by the poly data mapper can be chosen by simply picking the concrete implementation of the strategy.

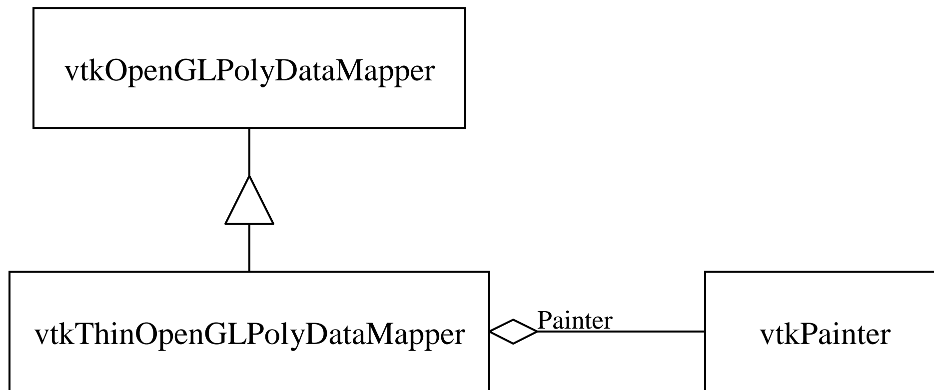

Figure 1 New poly data mapper.

Figure 1 shows a UML diagram of the layout for the new poly data mapper. From the outside, vtkThinOpenGLPolyDataMapper behaves like any other poly data mapper. It fits in the VTK pipeline, listens to render requests, and sets the OpenGL pipeline state based on things like the current lighting and material properties. It also draws OpenGL primitives, but not directly. Instead, this responsibility is delegated to a vtkPainter. The vtkPainter is responsible for nothing but issuing drawing commands. That is, it issues only OpenGL commands that send vertices and their properties into the pipeline (i.e. glVertex, glColor, glNormal, and so on) and commands that establish connectivity information (i.e. glBegin and glEnd).[2]

---

[1] *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.

[2] The painter may also issue these commands via other means such as vertex arrays or display lists.

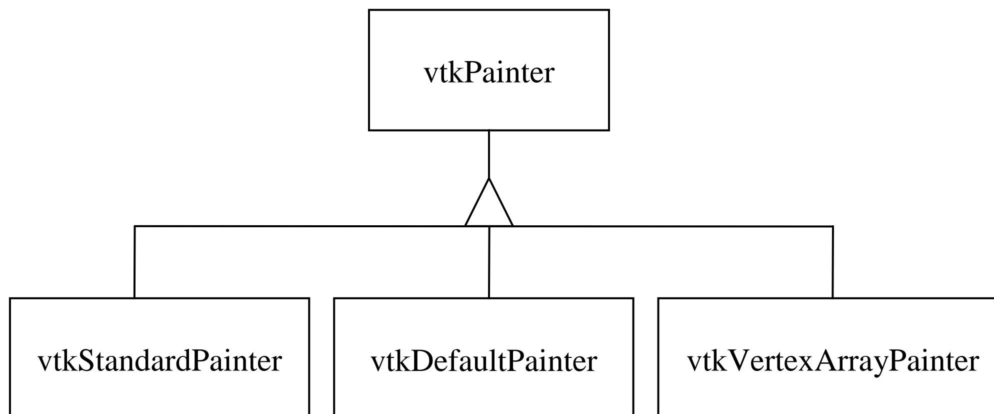

Figure 2 Sample implementations of vtkPainter.

The vtkPainter class is an abstract interface. Concrete subclasses must be instantiated to perform its functionality, as shown in Figure 2. An implementation of vtkPainter need only focus on one type of rendering. Furthermore, an implementation of vtkPainter need not be able to render every type of poly data. For example, vtkVertexArrayPainter may not work on poly data with cell data. Allowing a painter to focus on only a subset of possible poly data types should make the implementation easier to code, easier to read, and easier to optimize. Because implementing a new rendering scheme is as easy as subclassing vtkPainter, the poly data mapper becomes much more flexible.

3. vtkPainter

In this section we discuss the vtkPainter object and describe how it is implemented, subclassed, and used. First, we describe the interface of vtkPainter and provide the reasoning for this interface. Second, we describe a helper class that provides an abstract interface to the rendering system. Third, we list some straightforward implementations of vtkPainter that are currently available.

3.1. vtkPainter Interface

```
class vtkPainter : public vtkObject
{
public:
  vtkTypeRevisionMacro(vtkPainter, vtkObject);
  virtual void PrintSelf(ostream &os, vtkIndent indent);
  static vtkPainter *New();

  virtual void SetPolyData(vtkPolyData *arg);

  vtkBooleanMacro(StaticData, int);
  vtkBooleanMacro(ConserveMemory, int);
  vtkBooleanMacro(HighQuality, int);

  virtual void SetPointPosition(vtkDataArray *data);
  virtual void SetPointNormals(vtkDataArray *data);
  virtual void SetCellNormals(vtkDataArray *data);
  virtual void SetPointColors(vtkDataArray *data);
  virtual void SetCellColors(vtkDataArray *data);
  virtual void SetPointTCoords(vtkDataArray *data);
  virtual void SetCellTCoords(vtkDataArray *data);

  virtual void SetPointAttribute(int index, vtkDataArray *data);
  virtual void SetCellAttribute(int index, vtkDataArray *data);

  virtual void SetDevice(vtkPainterDeviceAdapter *d);

  virtual void DrawVerts(vtkRenderer *renderer);
  virtual void DrawLines(vtkRenderer *renderer);
  virtual void DrawPolys(vtkRenderer *renderer);
  virtual void DrawStrips(vtkRenderer *renderer);
...
```

Figure 3 Partial interface for vtkPainter.

Figure 3 lists a subset for an example listing of vtkPainter. For brevity, Figure 3 has several obvious omissions, such as the *Get* counterparts to all the *Set* methods. This section will discuss the methods included in the interface and detail why they are important.

The vtkPainter has the methods standard on all vtkObject classes, declared with vtkTypeRevisionMacro and PrintSelf. The New method returns a default implementation of vtkPainter (as described in Section 4).

A vtkPainter needs polygon data to render. The obvious storage format is a vtkPolyData object. The vtkPainter may also build a secondary representation for speedier rendering. Keeping a reference to the vtkPolyData object should not waste memory since the data will undoubtedly also be held elsewhere.

A vtkPainter may need to make choices as to the best way to render data. Thus, it holds some additional state as to the nature of the poly data and the rendering system being used. The Boolean flag StaticData gives the frequency that the

input data changes. If true, then slow preprocessing for fast frame rates will probably be rewarded. If false, then the preprocessing may occur too frequently to be worthwhile. The ConserveMemory flag signals whether the vtkPainter should avoid building large auxiliary structures, even at the expense of speed. The HighQuality flag signals whether the vtkPainter can make changes to the data that adversely affect image quality but may speed up rendering (for example turning cell data into point data).

Although the vtkPainter accepts a vtkPolyData object, this object is only used for connectivity information. Attribute information (*i.e.* point locations, normals, colors, and texture coordinates) are set explicitly. Attribute information is given as vtkDataArrays. Some arrays may come directly from attribute information in the poly data. Since this information is also stored as vtkDataArrays, the only overhead is a reference copy. Other arrays, such as the colors, will need to be built.

The attribute arrays can be defined over points or over cells (but not both). The one exception is the point position array, which, by definition, can only be defined over points.

There is also another convention of setting attribute arrays as unnamed, indexed arrays. These are useful when using a shading language such as Cg or the upcoming OpenGL 2.0 GLSL. When using a shader, the naming of the attribute may be meaningless. The number of attributes can also vary. The use of vtkPainter with high level shading languages is discussed in more detail in Section 6.

Table 1 Attribute aliasing in vtkPainter.

Alias	Index	OpenGL Function
PointPositions	0	glVertex
Normals	2	glNormal
Colors	3	glColor
TCoords	8	glTexCoord

The attribute arrays for point positions, normals, colors, and texture coordinates are really just aliases for numbered attributes. Table 1 lists how these aliases are mapped to the attributes. This aliasing was established to agree with similar mappings used in OpenGL extensions such as GL_ARB_vertex_program, GL_-NV_vertex_program, and GL_NV_vertex_program2.

vtkPainter also hold a vtkPainterDeviceAdapter. The device adapter is used to send rendering commands to a rendering system (such as OpenGL). Like the unnamed attributes, the abstract device adapter's primary purpose is to support high level shading languages. A high level shader may require attribute information to be sent in a way that is inconsistent with the functions used without the shaders. The interface for vtkPainterDeviceAdapter is described in detail in Section 3.2.

Finally, the vtkPainter has methods to draw polygons. There are separate methods for drawing points, lines, polygons, and triangle strips. The reason for this is twofold. First, the painter will undoubtedly have different techniques to render each type and may in fact choose not to render some types. Second, the rendering parameters for each type are probably different, and having different methods allows the calling vtkThinPolyDataMapper to change the rendering state for each type.

3.2. vtkPainterDeviceAdapter Interface

```
class vtkPainterDeviceAdapter : public vtkObject
{
public:
  vtkTypeRevisionMacro(vtkPainterDeviceAdapter, vtkObject);
  virtual void PrintSelf(ostream &os, vtkIndent indent);

  virtual void BeginPrimitive(int mode) = 0;
  virtual void EndPrimitive() = 0;
  virtual void SendAttribute(int index, int components, int type,
                             const void *attribute) = 0;

  void SetAttributePointer(int index,
                           vtkDataArray *attributeArray);
  virtual void SetAttributePointer(int index, int numcomponents,
                                   int type, int stride,
                                   const void *pointer) = 0;
  virtual void EnableAttributeArray(int index) = 0;
  virtual void DisableAttributeArray(int index) = 0;
  virtual void DrawArrays(int mode, vtkIdType first,
                          vtkIdType count) = 0;
  virtual void DrawElements(int mode, vtkIdType count, int type,
                            void *indices) = 0;

  virtual int Compatible(vtkRenderer *renderer) = 0;
};
```

Figure 4 Interface for vtkPainterDeviceAdapter.

Figure 4 shows the interface for vtkPainterDeviceAdapter. The methods in vtk-PainterDeviceAdapter intentionally parallel the drawing functions available in OpenGL. There is one implementation of vtkPainterDeviceAdapter available: vtkOpenGLPainterDeviceAdapter. I will describe the interface of vtkPainter-DeviceAdapter in terms of the OpenGL implementation. This will make the interface clear for anyone reasonably familiar with OpenGL programming.

Like OpenGL 1.1, vtkPainterDeviceAdapter supports two modes of defining primitives: directly sending attributes and vertex arrays. The first mode demarcates the primitives with the BeginPrimitive and EndPrimitive methods, which have the same function as glBegin and glEnd. The attributes for each vertex are established with the SendAttribute method. The OpenGL implementation of SendAttribute calls one of the glVertex, glColor, glNormal, or glTexCoord functions. The conversion from attribute number to OpenGL function is consistent with the vtkPainter aliasing and is given in Table 1. As attributes are specified, a current set of attributes is mainatined. When SendAttribute method is called for attribute 0, all attributes are captured for the vertex and sent to the rendering system. This is consistent with the behavior of glVertex.

The second mode for sending primitives uses vertex arrays. Instead of sending attributes one at a time, the application can store groups of attributes in arrays and then instruct the vtkPainterDeviceAdapter to send many of these attributes at once. Before vertex arrays may be used, the attribute pointers must be established. This is done with the SetAttributePointer methods. They correspond to the glVertexPointer, glColorPointer, glNormalPointer, and glTexCoordPointer functions. The mapping from attribute number to OpenGL function is again consistent with Table 1. The attribute arrays in play are selected with the EnableAttributeArray and DisableAttributeArray methods, which correspond to the glEnableClientState and glDisableClientState functions. The attributes are sent to the rendering system when DrawArrays or DrawElements is called. As part of their arguments, the order to send the attributes is specified. These two methods correspond to the glDrawArrays and glDrawElements functions.

One final feature of vtkPainterDeviceAdapter is that it can report whether it can be used with a particular vtkRenderer with its Compatible method.

3.3. Some vtkPainter Implementations

Here is a list of some straightforward implementations of vtkPainter. That is, these implementation directly draw the poly data they are given. Sections 4 and 5 list some other vtkPainter implementations with different behavior.

- **vtkStandardPainter** is jack of all trades and a master at none. This is a very straightforward implementation of a vtkPainter that can handle any type of vtkPainter. However, little effort is made in optimizing the speed of the rendering (in particular, the rendering is seriously API bound). The vtkStandardPainter can be used as catch-all for all types of rendering although an alternative method should be used if found.

- **vtkVertexArrayPainter** reduces the API overhead by using vertex arrays. However, vtkVertexArrayPainter does not work with any poly data that has attributes defined over cells (point data only).

- **vtkOpenGLVertexArrayPainter** is like vtkVertexArrayPainter but it also uses OpenGL extensions for vertex array objects and primitive restarts.

- **vtkStreamingPainter** iterates over attributes in much the same way as vtkStandardPainter but caches the values in vertex arrays rather than sending them directly to the vtkPainterDeviceAdapter.

4. Choosing a vtkPainter

The main disadvantage of presenting users with a choice is that they now have to choose. Because there was only one poly data mapper, users did not have to choose one. However, now users may be faced with perhaps many painters. How will a user which one to pick? How can a user know which painter will be best at drawing the data on the current platform? To make matters more complicated, not all painters will be capable of performing the drawing. There may also be painters of which the user is unaware.

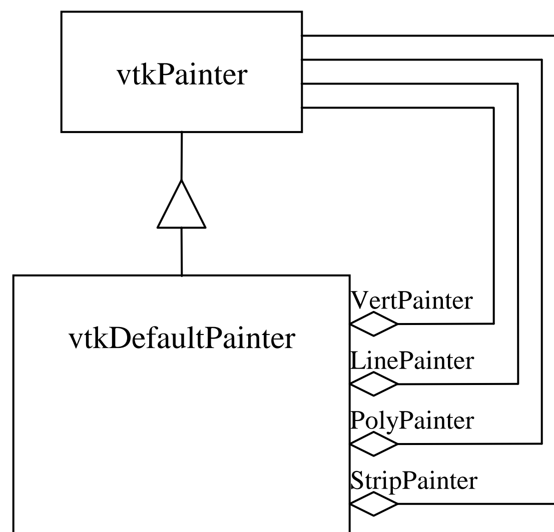

Figure 5 The painter chooser object.

The solution is to create another object, called vtkDefaultPainter, that knows about a set of painters and their capabilities and choose one for the user. The vtkDefaultPainter object inherits from vtkPainter, as shown in Figure 5, so that it looks like any other painter to the rest of the program. However, vtkDefaultPainter does not itself draw anything. Instead, using the poly data, rendering system, and internal flags as criteria, chooses the best fit painters. The drawing is then delegated to these painters.

If an application has no reason to pick a particular painter, it simply uses vtk-DefaultPainter as a default. That way, the "best" painter will always be chosen. Of course, there may be painters of which vtkDefaultPainter is not aware. However, when someone writes a new painter or set of painters, they also have the option to make a new painter chooser. She could then register her painter chooser with the vtkObjectFactory. In this way, a user can plug a custom set of painters into an application without modifying the application.

5. Code Reuse

Another issue with this approach is that of code reuse. A good deal of code may be replicated amongst the painters, whereas a monolithic class such as vtkOpen-GLPolyDataMapper can easily reuse its own code. For example, what if we wanted a painter that built display lists? We would still need code to draw the primitives. Rather than re-implement the drawing code from, say, vtkStandard-Painter, we would prefer to reuse it.

Some amount of code replication is inevitable, but there are some approaches that will drastically reduce the amount of code replication. One method is simply object inheritance. That is, one object may be able to reuse a significant amount of code from another object by incorporating that object as part of itself. Although we shall certainly use inheritance in this way in the vtkPainter hierarchy, the technique for use in code reuse is limited. Trying to implement all of our code reuse in this will lead to obfuscated code (case in point, see vtkOpen-GLVertexArrayPolyDataMapper and its subclass vtkNVidiaPolyDataMapper).

For more code reuse, we again go to the gang-of-four book. This time we apply the *Decorator* pattern. In the *Decorator* pattern, a component delegates most of its work to another component. However, it also modifies the result in some way before or after delegating the work.

For example, consider the display list painter. The display list painter can compile its display lists by calling on another vtkPainter to do the actual drawing. *Viola*, we have any type of rendering mode with the added feature of non immediate mode rendering.

The *Decorator* pattern gives this design a big advantage. The decorator painters can be combined in weird and wonderful ways. Here are the currently available decorators (and some ideas for decorators).

- **vtkOpenGLDisplayListPainter** compiles and renders display lists as described previously.

- **vtkSortingPainter** reorders the polygons based on the camera direction.

- **vtkFilterPainter** applies a filter to the polygon data (*not implemented*).

- **vtkViewDependentLODPainter** applies a view dependent level of detail to the polygon data (*not implemented*).

6. Using vtkPainter with Shaders

The current generation of graphics hardware has a high degree of programmability. A user level application can replace the vertex and fragment processing units of their cards with their own programs. This programmability lends to a much more powerful rendering process.

Despite the flexibility of the pipeline, the data is sent to the card in much the same way. Data are sent as attributes on a per-vertex basis. Thus, the algorithms we develop within vtkPainter should also be used to drive modified graphics pipelines.

Although the mechanism is the same, the actual functions used to send attribute information changes when using vertex programs. This was the primary motivating factor for creating the vtkPainterDeviceAdapter (see Section 3.2). The vtkPainterDeviceAdapter replaces the OpenGL calls used to send primitives to the graphics system. Because they all use this abstract interface, you can use vtkPainters with vertex and fragment programs by subclassing vtkPainterDeviceAdapter to call the appropriate functions.

Note that the number of attributes specified per vertex is limited only by the underlying implementation of the graphics shader. Any attribute that is not classified as point position, color, normal, or texture coordinate can simply be added as an unnamed attribute.

7. Concluding Remarks

The vtkPainter hierarchy provides a powerful and flexible architecture for implementing mapping in VTK. Separating the drawing algorithm from the mapper makes it easier to implement more efficient drawing algorithms. It also allows the drawing algorithm to be selected and replaced dynamically without *a-priori* knowledge. Using decorators, we can modify the behaviors of our drawing algorithms. We can mix and match behaviors to provide unique and powerful mappings. By having an abstract device interface, most painters are independent of the rendering system used. They can be applied to graphics pipelines with programmed units or different rendering APIs altogether.

The only major disadvantage of the vtkPainter hierarchy is the added overhead of the more generic features. The abstract vtkPainterDeviceAdapter means a higher overhead for each function call. The limitless attributes mean iterating over all attribute types. The looser coupling between decorators and the algorithm they modify mean more memory reordering and copying. However, the flexibility of

the system should ensure that, in general, a better algorithm for mapping is being used. The appropriate use of rendering resources should mitigate any added overhead vtkPainter imposes.

8. Acknowledgements

This work was done at Sandia National Laboratories. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.