Generic Vertex Attributes

KITWARE INC.

December 12, 2007

1 Introduction

OpenGL provides mechanisms to pass parameters that as associated with each vertex eg. normal, position etc. These are referred to as *standard vertex attributes*. Besides these vertex attributes that are interpreted by the default rendering pipeline, OpenGL also provides mechanisms to pass application defined parameters known as *generic vertex attributes* or simply as *generic attributes*. The number of generic attributes is implementation dependent. The default rendering pipeline has no knowledge to convert user defined primitives into rendering parameters, hence they are generally ignored except when using programmable vertex shaders. In that case the generic attributes are available to the shaders just as standard attributes and the shader can use them to manipulate the properties of the rendered vertex. Vertex shaders written in both OpenGL standard GLSL or NVidia's Cg have access to generic attributes. In Cg terminology generic attributes are often referred to as *varying parameters* i.e. parameters that change with each vertex.

This document describes the use and implementation of the extension added to VTK to support passing of generic vertex attributes.

2 Thanks

Support for generic vertex attributes in VTK was contributed in collaboration with Stephane Ploix at EDF.

3 Usage

This extension is available at the head of the VTK CVS repository since Oct 04, 2007.

vtkPainterPolyDataMapper is the default polydata mapper used by VTK i.e. when user creates an instance of vtkPolyDataMapper, the object factory returns a vtkPainterPolyDataMapper instance. This mapper now supports api to map data arrays from dataset attributes to generic attributes identified by some name in the shader program.

```
void MapDataArrayToVertexAttribute(const char* vertexAttributeName,
    const char* dataArrayName, int fieldAssociation, int componentno=-1)
```

where:

- *vertexAttributeName* is the name with which the attribute is referred to in the shader program.
- *dataArrayName* is the name of the data array to be mapped to the attribute.
- *fieldAssociation* indicates if the data array is a point data array or a cell data array *vtkDataObject::FIELD_ASSOCIATION_POINTS* or *vtkDataObject::FIELD_ASSOCIATION_CELLS* respectively.
- *componentno* indicates which component of the data array must be passed to the rendering pipeline. If -1, all components are passed. OpenGL supports passing up to 4 components per attribute.

One can add several attribute mappings however the vertexAttributeNames must be unique.

To remove mappings, following API is provided:

```
void RemoveVertexAttributeMapping(const char* vertexAttributeName);
void RemoveAllVertexAttributeMappings();
```

Note that only those vertex attributes that are referred to in the vertex shader will be passed on to the rendering pipeline. If no shader is loaded, none of the attributes are passed.

4 Implementation

When user uses MapDataArrayToVertexAttribute on the mapper to add mappings, the vtkPainterPolyDataMapper updates an instance of vtkGenericVertexAttributeMapping. vtkGenericVertexAttributeMapping is used to simply store the mapping. This vtkGenericVertexAttributeMapping instance is passed to the painters through the PainterInformation object using key

`vtkPolyDataPainter::DATA_ARRAY_TO_VERTEX_ATTRIBUTE.`

When this key is present in the information and a shader is loaded, the vtkPrimitivePainter, which is an abstract painter for implementing fast path rendering of different primitives, sets a flag to indicate to its subclasses that generic attributes are to be passed. The fast path painters exploit the information about what standard attributes are being passed to the rendering pipeline to avoid conditionals in the loop that iterates over cells/points. That being the case, the performance benefits of avoiding loops in the fast pass implementations cannot be exploited and hence all the fast path implementations i.e.

vtkPrimitivePainter subclasses, are skipped. As a result, the rendering responsibility is offloaded to vtkStandardPolyDataPainter. This painter iterates over all cells and points and passes standard vertex attributes as well as generic vertex attributes to the rendering pipeline.

For passing the standard vertex attributes, vtkStandardPolyDataPainter uses the vtkPainterDeviceAdapter subclass which is graphics system specific. Similarly for passing the generic vertex attributes, it uses the vtkShaderDeviceAdapter subclass which is shading language specific. We provide two implementations for GLSL (vtkGLSLShaderDeviceAdapter) and Cg (vtkCgShaderDeviceAdapter). Since this depends on the shading language used, the painter obtains this adapter from the the shader program loaded on the vtkProperty.

The crux of the API supported by vtkShaderDeviceAdapter is

```
void SendAttribute(const char* attrname, int components, int type,
  const void* attribute, unsigned long offset=0)
```

where:

- *attrname* identifies the name of the generic attribute.
- *components* gives the number of components in the attribute.
- *type* is a VTK type enumeration (VTK_FLOAT, VTK_INT etc.). A rendering system may not support all types for the attribute.
- *attribute* is the actual data for the attribute.
- *offset* if specified, is added to the attribute pointer *after* it has been casted to the proper type.

5 Examples

Two tests are provided in the VTK suite that test Cg and GLSL generic vertex attributes. These are:

1. TestGenericVertexAttributesGLSL
   (VTK/Rendering/Testing/Tcl/TestGenericVertexAttributesGLSL.tcl)
2. TestGenericVertexAttributesCg
   (VTK/Rendering/Testing/Tcl/TestGenericVertexAttributesCg.tcl)

These tests serve as good examples for this new functionality. Following are the python version for the same.

5.1 TestGenericVertexAttributesGLSL Python

```
import vtk

xmlMaterial = """<?xml version="1.0" encoding="UTF-8"?>
<Material name="GenericAttributes1">
  <Shader
    scope="Vertex"
    name="VertexShader"
    location="Inline"
    language="GLSL"
    entry="main">
    attribute vec3 genAttrVector;
    varying vec4 color;

    void main(void)
    {
      gl_Position = gl_ModelViewProjectionMatrix *gl_Vertex;
      color = vec4(normalize(genAttrVector), 1.0);
    }
  </Shader>

  <Shader scope="Fragment" name="FragmentShader" location="Inline"
    language="GLSL" entry="main">
    varying vec4 color;
    void main(void)
    {
      gl_FragColor = color;
    }
  </Shader>
</Material>"""

renWin = vtk.vtkRenderWindow()
iren = vtk.vtkRenderWindowInteractor()
iren.SetRenderWindow(renWin)

renderer = vtk.vtkRenderer()
renWin.AddRenderer(renderer)

src1 = vtk.vtkSphereSource()
src1.SetRadius(5)
src1.SetPhiResolution(20)
src1.SetThetaResolution(20)

randomVectors = vtk.vtkBrownianPoints()
randomVectors.SetMinimumSpeed(0)
randomVectors.SetMaximumSpeed(1)
randomVectors.SetInputConnection(src1.GetOutputPort())
```

```
mapper = vtk.vtkPolyDataMapper()
mapper.SetInputConnection(randomVectors.GetOutputPort())

actor = vtk.vtkActor()
actor.SetMapper(mapper)

# Load the material. We are loading the material
# described in the string.
actor.GetProperty().LoadMaterialFromString(xmlMaterial)

# Set red color to show if shading fails.
actor.GetProperty().SetColor(1.0, 0, 0)

# Turn shading on. Otherwise, shaders are not used.
actor.GetProperty().ShadingOn()

# Map PointData.BrownianVectors (all 3 components)
# to genAttrVector
mapper.MapDataArrayToVertexAttribute(
  "genAttrVector","BrownianVectors",0, -1)

renderer.AddActor(actor)
renderer.SetBackground(0.5, 0.5, 0.5);
renWin.Render()

renderer.GetActiveCamera().Azimuth(-50)
renderer.GetActiveCamera().Roll(70)

iren.Start();
```

5.2 TestGenericVertexAttributesCg Python

```
import vtk

xmlMaterial = """<?xml version="1.0" encoding="UTF-8"?>
<Material name="GenericAttributes1">
  <Shader
    scope="Vertex"
    name="VertexShader"
    location="Inline"
    language="Cg"
    entry="main">
      <MatrixUniform name="ModelViewProj"
        type="State"
        number_of_elements="2"
        value="CG_GL_MODELVIEW_PROJECTION_MATRIX CG_GL_MATRIX_IDENTITY" />
```

```
      <MatrixUniform name="ModelViewIT"
        type="State"
        number_of_elements="2"
        value="CG_GL_MODELVIEW_MATRIX CG_GL_MATRIX_INVERSE_TRANSPOSE" />

      struct appin
      {
          float4 Position : POSITION;
          float3 Normal   : NORMAL;
      };

      // define outputs from vertex shader
      struct vertout
      {
          float4 HPosition : POSITION;
          float4 Color0    : COLOR0;
      };

      vertout main(appin IN,
                   uniform float4x4 ModelViewProj,
                   uniform float4x4 ModelViewIT)
      {
        vertout OUT;

        // transform vertex position into homogenous clip-space
        OUT.HPosition = mul(ModelViewProj, IN.Position);

        OUT.Color0.xyz = normalize(IN.Normal);
        OUT.Color0.a = 1.0;
        return OUT;
      }
  </Shader>
</Material>
"""


renWin = vtk.vtkRenderWindow()
iren = vtk.vtkRenderWindowInteractor()
iren.SetRenderWindow(renWin)

renderer = vtk.vtkRenderer()
renWin.AddRenderer(renderer)

src1 = vtk.vtkSphereSource()
src1.SetRadius(5)
src1.SetPhiResolution(20)
src1.SetThetaResolution(20)

randomVectors = vtk.vtkBrownianPoints()
randomVectors.SetMinimumSpeed(0)
```

```
randomVectors.SetMaximumSpeed(1)
randomVectors.SetInputConnection(src1.GetOutputPort())

mapper = vtk.vtkPolyDataMapper()
mapper.SetInputConnection(randomVectors.GetOutputPort())

actor = vtk.vtkActor()
actor.SetMapper(mapper)

# Load the material. Here, we are loading a material
# defined in the Vtk Library. One can also specify
# a filename to a material description xml.
actor.GetProperty().LoadMaterialFromString(xmlMaterial)

# Set red color to show if shading fails.
actor.GetProperty().SetColor(1.0, 0, 0)

# Turn shading on. Otherwise, shaders are not used.
actor.GetProperty().ShadingOn()

# Map PointData.BrownianVectors (all 3 components) to genAttrVector
mapper.MapDataArrayToVertexAttribute("IN.Normal","BrownianVectors",0, -1)

renderer.AddActor(actor)
renderer.SetBackground(0.5, 0.5, 0.5);
renWin.Render()

renderer.GetActiveCamera().Azimuth(-50)
renderer.GetActiveCamera().Roll(70)

iren.Start();
```
