11.4-3

OpenVG Paint Subsystem over OpenGL ES Shaders

Mathieu Robart, AST
STMicroelectronics, Bristol, UK

***Abstract*—OpenVG is the 2D vector graphics API developed by Khronos. Full hardware acceleration of this API is not yet widely supported, unlike its 3D companion OpenGL ES. As a consequence, using ES compliant graphics hardware as a support for accelerating OpenVG comes naturally.**

In this paper, we summarize the four paint modes supported by OpenVG (color, linear gradient, radial gradient and pattern) and propose their possible implementation as GLSL ES fragment shaders running over an OpenGL ES compliant 3D graphics pipeline.

We conclude with a proposal for enhancing the graphics hardware in order to support complex color ramps as supported by OpenVG.

I. Introduction

OpenVG is a 2D vector graphics API developed by the Khronos Group[1], who are responsible for the popular OpenGL ES 2.0 programmable 3D graphics API [2].

Both OpenVG and OpenGL ES 2.0 models rely on pipeline architectures, where an abstract description of the object to draw (e.g. 2D Bezier path for OpenVG, 3D triangularized model for OpenGL) is transformed, then rasterized as pixels to be displayed on the screen after further tests and color manipulations (see Figure 1 for a simplified representation of both pipelines).

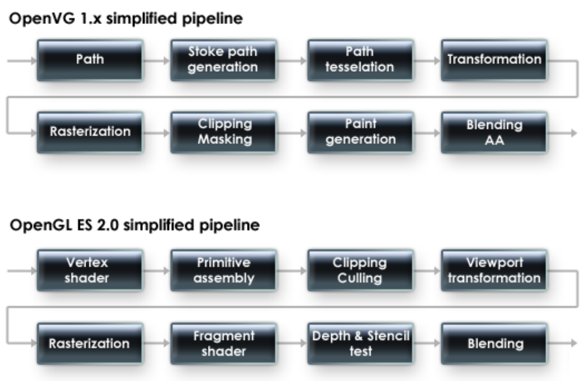

Figure 1: OpenVG and OpenGL ES pipelines

Hardware acceleration support is beginning to be widely available for OpenGL ES 2.0 in the form of GPUs. These processors are programmable at the vertex and pixel level, and this flexibility offers a new perspective in terms of computation. Indeed, these GPUs can now be used to perform tasks for which they were not originally intended, for example the acceleration of other APIs such as OpenVG.

This paper discusses more precisely a possible implementation of the Paint Generation stage, a particularly compute intensive stage of the OpenVG pipeline. The approach is effective for a programmable GPU compliant with OpenGL ES 2.0.

OpenVG supports the drawing of smooth, resolution-independent 2D shapes described as mathematical paths (e.g. Bezier curves, lines, and ellipses). These shapes can be rendered as a simple single stroke, or with their inside area filled by a user-defined paint. Four paint modes are defined (see Figure 2). The simplest mode is Color paint, where each pixel uses the same fixed color and alpha. Gradient paint sets each pixel to a different color according to a linear or radial interpolation of a color ramp. Finally, pattern paint defines each pixel color according to a predefined pattern image (similar to a texture in 3D graphics).

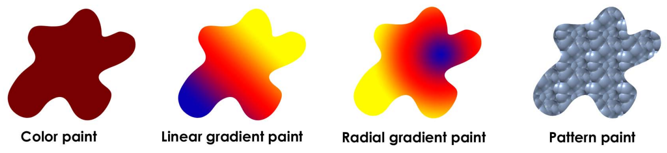

Figure 2: Different OpenVG paint modes

This paper presents these different paint modes in more detail, and proposes a possible implementation of each of them as an ESSL fragment.

For simplicity, we have only considered the pad spread mode of the gradient function. Other spread modes include *repeat* and *reflect*.

II. Implementation of Paint Modes

A. *Color paint*

In this mode, each pixel takes a unique color and alpha value, according to the current color stored in the context. This case is trivial, and the fragment shader is:

```
uniform vec4 solidColor;
void main ()
{
   gl_FragColor = solidColor;
}
```

The fragment takes the default color, passed as an uniform, and assigns it to the final color of the pixel.

B. *Linear and Radial gradient paint*

With these paint modes, the color of each pixel is determined by interpolating a fixed color ramp according to a gradient function depending on the paint mode and on the location of the pixel on screen.

The linear gradient function takes the value 0.0 for all pixels located along the line orthogonal to the linear interpolation direction, passing by (x_0, y_0) (see Figure 3). It takes the value

1.0 for all pixels located along the line orthogonal to the linear interpolation direction, passing by (x_1, y_1).

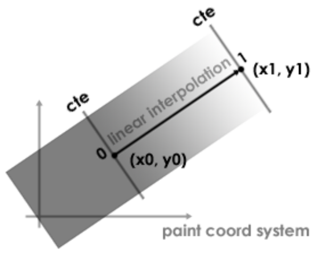

Figure 3: Linear gradient geometry

The linear gradient paint computes an offset, and a color is interpolated from a color ramp associated to the current paint. The corresponding shader is:

```
uniform vec4  gradP;  // gradient control points
uniform float gradDenom, colorStopsOffsets[7];
uniform float colorStopsInvInterval[6];
uniform vec4  colorStopsColors[7];
uniform int   colorStopsCountMinusOne;

void main ()
{
   float g = (gradP.z*(gl_FragCoord.x - gradP.x) +
gradP.w*(gl_FragCoord.y- gradP.y)) * gradDenom;

   if (g >= 1.0)
      gl_FragColor=
colorStopsColors[colorStopsCountMinusOne];
   else
   {
      for(int i=0; i<colorStopsCountMinusOne; i++)
         if (g < colorStopsOffsets[i])
         {
           float  coef  =  (colorStopsOffsets[i]-g)  *
colorStopsInvInterval[i];
           gl_FragColor= mix (colorStopsColors[i+1],
colorStopsColors[i], coef);
            break;
         }
   }
}
```

The radial gradient paint is defined in a very similar way to the linear gradient paint, but using a different formulation for the gradient equation (see [3]).

The corresponding shader is:

```
uniform vec4  gradP; // gradient control points
uniform float gradDenom, gradRadius2;
uniform float colorStopsOffsets[7]
uniform float colorStopsInvInterval[6];
uniform vec4  colorStopsColors[7];
uniform int   colorStopsCountMinusOne;

void main ()
{
   vec2 dxy = gl_FragCoord.xy - gradP.xy;
   float g  = dot(dxy,  gradP.zw);
   float h  = dxy.x * gradP.w – dxy.y * gradP.z;
   g += sqrt(gradRadius2 * dot(dxy, dxy) - h*h);
   g *= gradDenom;

   if (g >= 1.0)
   [… see linear gradient code for remaining code]
}
```

C. *Pattern paint*

This paint mode is similar to texture mapping. However, texture coordinates are not used. Instead, an inverse transformation is used: the inverse of the matrix corresponding to the multiplication of the paint-to-user and user-to-surface transformation matrices.

This matrix can be passed as a uniform variable to the fragment shader, and a 3D matrix multiplication is then necessary per fragment. The corresponding shader can be written as:

```
uniform sampler2D  sampler;
uniform mat3       invTransfo;

void main ()
{
  vec3 uvw;

  uvw = invTransfo * vec3(gl_FragCoord.xy, 1.0);
  gl_FragColor = texture2D(sampler, uvw.xy);
}
```

III. Hardware improvements

An external mechanism could be added (similar to a simple texture unit) for handling the color ramp interpolation.

More generally, each color stop uses up to 5 floating-point numbers: one offset between 0.0 and 1.0, and a RGBA color possibly coded over 4 floating-point numbers (one for each channel). The OpenVG specification requires support for at least 32 color stops, so a local memory of 640 bytes is enough for storing them at full precision. Less memory could be necessary if less precision is judged sufficient.

The look-up mechanism, performed according to the gradient value and interpolation between the two corresponding color stops, could be accomplished instead in hardware. Such an extension to the ES 2.0 graphics pipeline is under development and should provide a way to hide the potentially high latency of the color ramp interpolation process, leaving precious cycles to the fragment shading unit for proper shading code execution.

IV. Conclusion

We have presented in this paper how the different paint modes supported by OpenVG can be mapped to a programmable OpenGL ES 2.0 compliant architecture and we have provided the corresponding ESSL shader codes.

OpenGL ES hardware accelerators are becoming widely available in various devices and the reuse of these programmable assets for accelerating other APIs such as OpenVG is becoming increasingly necessary.

Reference

[1] Khronos group, www.khronos.org
[2] A. Munshi. OpenGL ES, Common Profile Specification 2.0. Khronos group September 2007.
[3] D. Rice, OpengVG Specification Version 1.0.1. Khronos group, March 2007.