SharpMedia Vector Graphics Design

© 2007 Bojan Šernek and Žiga Osolin – SharpMedia project, document version: DRAFT

Table Of Contents

[About 2](#_Toc177348937)

[Interfaces 2](#_Toc177348938)

[IFill 2](#_Toc177348939)

[ITransform 2](#_Toc177348940)

[IPen 2](#_Toc177348941)

[ICanvas 2](#_Toc177348942)

[Implementation Notes 4](#_Toc177348943)

[Usage Case: Drawing a HUD 5](#_Toc177348944)

# About

The SharpMedia Vector Graphics library (the library) is designed as a wrapper around the SharpMedia Graphics library. It enhances the graphics library with resolution independent vector graphics.

In the current DRAFT state of this document, there are important features missing:

* Fonts and text
* Classes and static methods that add to the implementation notes

# Interfaces

## IFill

An **IFill** describes the fill used. A fill specifies the appearance of object. Examples of fills are **SolidFill, GradientFill, RadialFill, TextureFill …** Each fill must provide information that are necessary for **ICanvas** to construct a shader. This is done by making fill a shader interface that can be used through **FillElement** element defined by Vector Graphics.

## ITransform

The ***ITransform*** interface represents a geometry transform. If it is linear, it must provide a valid converter to 4x4 matrices. It must expose a method that transforms the incoming untransformed geometry to the transformed one. Through this method, non-linear transforms can be applied to geometry.

## Pen

A pen is an object for drawing outlines. It is essentially a fill with additional properties. Those properties include line width, ending type and anti aliasing. The fill is used to actually render the line while other information is used to tessellate the line to triangles.

## ICanvas

The library is built on top of the ***ICanvas*** interface, which is an abstraction of a three dimensional space projected to a two dimensional plane. This plane is fixed such that it views the “scene” along the z axis of the virtual three dimensional spaces.

A canvas is a thread safe renderer that receives orders and executes them in a timely fashion. The rendering code is executed between **Begin()/End()** calls. This also makes canvas thread safe. **Begin()/End()** can be nested as long as there is one **End()** call for each **Begin()** call. Only the outer call locks the device. Rendering state (transform, pixel density …) is pushed on **Begin()** and popped on **End().**

The canvas can be queried for pixel density (conversion from virtual units used in resolution independent rendering it performs to numbers of pixels on output device), can transform incoming virtual unit coordinates to projected 2D pixel coordinates on the output device, can get or set the amount of tessellation that is performed, in relativity to the pixel count.

Rendering is issued with ***Draw*** or **Fill** methods. The **Draw** method draws the outline of any **IPath** object given an **IPen** interface. A **Fill** method fills the shape. The shape must be at least **ITesselatable**. Other shape interfaces and properties may also be used by vector graphics library. Refer to specific canvas specifications for them.

Canvas also supports events on specific actions: line drawn, shape drawn and flushed event. Those events can customize lines or shapes before drawing.

A canvas is bound to a device that it uses in shared way (it enters when rendering and exits when rendering is complete).

# Implementation Notes

Canvas should provide many implementations based that work best with different types and quantity of vector data.

1. **NullCanvas** –does not render.
2. **GraphicsCanvas** – an implementation of canvas that draws data in the same order as data is given without the use of Z-buffers. Used when non-commutative alpha blending is needed or where caching does not give better performance.
   1. Implementation should tessellate all objects to triangles and push them to current buffer;
   2. Implementation creates a complex shader (at construction time) from all given fill types. Some fill interfaces may be reused (if no global data is needed) while some may be reimplemented many times.
   3. On End() or when too much data is in the buffer or all slots for specific fill are used (for example, **TextureFill** type is reimplemented 10 times and all 10 texture slots are full), data is flushed to **GraphicsDevice**.
3. **SubCanvas** – redirects data to parent Canvas. Before that, it performs clipping and proper scalling/translation.
4. **TextureCacheCanvas –** implementation should be based on **GraphicsCanvas**. It can save some “layers” as textures and reuse them on next render if not changed (this is doubtful to improve performance).
5. **ReorderCacheCanvas** – the implementation saves all the geometry and sorts them by their fill. This way, data with the same fill are rendered together. In order to allow this, Z-buffer must be used.
6. **GSCanvas** - acts like a graphics canvas but tessellates geometry (built-in geometry) in geometry shader. Should give a performance boost if a lot of curves lines/surfaces are used because GS tessellation is faster.
7. **GSReorderCanvas** - combines geometry shader tessellation and reordering of geometry with the use of Z-buffer.

Fill is implemented as interface to **FillElement**. The element switches between fill ids and chooses the correct id. Then the fill is executed, producing colour. Each fill is given the following attributes:

* The position of pixel (normalized coordinates);
* Distance from nearest end of shape (normalized coordinates);
* Texture coordinate (2D);
* Custom attribute(s) – the maximum is implementation defined (so not all fills may be used in conjuction with all canvases);

A pen is only an extension of full to provide meaningful information for rendering lines.

**Scenario: Rendering Shapes**

All shapes are rendered as collection of triangles. Shape is tessellated (either in CPU or by GS) to triangles and they are pushed further down the pipeline. The tessellation density is specified to canvas and it must obey it.

**Scenario: Rendering Lines**

Lines have thickness. We also render them as such, using two triangles (a quad). This is to allow lines and shapes to share the same fill and to be packed in the same buffer. End and start caps are rendered as circles.

# Usage Case: Drawing a HUD

Drawing a simple heads up display, using 2D points:

public class HUD : ICanvasRenderable

{

private Vector2f[] enemyLocations;

private IFill usedFill;

static IFill redFill = null;

static IFill blueFill = null;

static Vector2f topLeftOffset = new Vector2f (-0.05, -0.05);

static Vector2f botRightOffset = new Vector2f ( 0.05, 0.05);

static double playerHealth = 100.0;

public void Render(ICanvas canvas)

{

if (playerHealth < 20.0)

{

usedFill = redFill;

}

else

{

usedFill = blueFill;

}

foreach (Vector2f position in enemyLocations)

{

canvas.Fill(new Rectf(new Vector3f(position - topLeftOffset, 0),

new Vector3f(position + botRightOffset, 0),

usedFill, null));

}

}

… the class continues …

}

In this example, the HUD rendering code is oblivious of the actual pixel size of the 0.05 units of space. The units of space may therefore be specified physically, in centimeters or any other physical unit of length and then guaranteed through the rendering interface to map to appropriate pixel size. We could ask users about their screen diagonal size in inches for example.