TinyLine 2D Programming Guide

Version 2.5

Latest version: http://www.tinyline.com/

Change history

November 4, 2006	Version 1.11	Initial document release
November 26, 2007	Version 2.0	Updated to cover 2.0 API
November 16, 2008	Version 2.1	Updated to cover 2.1 API
February 6, 2010	Version 2.3	Updated to cover 2.3 API
April 27, 2012	Version 2.5	Updated to cover 2.5 API

Table of Content

TinyLine 2D Programming Guide 1
Table of Content 3
1. TinyLine 2D Overview 5
1.1 Introduction 5
1.2 Key features 5
1.3 The TinyLine 2D API 5
1.4 Drawing pipeline 7
1.5 The painters model 8
1.6 Coordinate spaces 8
1.7 Graphics elements 8
1.8 Text and fonts 8
1.9 Colors 8
1.10 TinyLine 2D Examples 9
2. Fixed Point Numbers 10
3. Coordinate Spaces and Transformations 11
3.1 Device space 11
3.2 User space 11
3.3 Character space 12
3.4 Relationships among coordinate spaces 12
3.5 Transformations 12
4. Graphics State - TinyState 15
4.1 Antialiasing 16
4.2 Background color 17
4.3 Line cap style 17
4.4 Line dash pattern 19
4.5 Shape bounds 20
4.6 Fill Alpha 20
4.7 Fill Color 21
4.8 Fill Rule 21
4.9 Global Alpha 22
4.10 Line join style 23
4.11 Matrix 24
4.12 Miter limit 24
4.13 Smooth bits 25
4.14 Stroke Alpha 26
4.15 Stroke Color 26
4.16 Stroke Width 26
4.17 Text layout direction 26
5. Paths and shapes 27
5.1 Paths 27
5.2 Basic Shapes 30
5.3 Bounds and hit test 32
6. Text and Fonts 34

6.1 Introduction 34
6.2 Fonts and Glyphs 34
6.3 Bounds and hit test 36
6.4 Measuring text before drawing 37
7. Colors 39
7.1 Introduction 39
7.2 Single Color 39
7.3 Gradients 40
7.4 Stops colors 40
7.5 Spread method 40
7.6 Gradient Units and Gradient transform 41
7.7 Linear Gradient 41
7.8 Radial Gradient 41
7.9 Patterns 43
8. References 46

1. TinyLine 2D Overview

1.1 Introduction

TinyLine 2D implements a mobile 2D graphics engine for Java platform. TinyLine 2D handles basic shapes, paths, texts, outlined fonts and images in a uniform way. The TinyLine 2D provides access to powerful features such as transparency, path-based drawing, offscreen rendering, advanced color management, antialiased rendering.

Being pure Java based, TinyLine 2D provides the unified 2D graphics engine for a vast variety of Java platforms and profiles. As a result, developers are easily able to incorporate high quality, scalable and platform-independent graphics into their Java applications across different Java platforms.

1.2 Key features

- Small footprint
- Fast fixed-point numbers mathematics
- Paths, basic shapes and texts drawings
- Hit tests for paths and texts
- Solid color, bitmap, pattern, gradient (radial and linear) paints
- Fill, stroke and dash
- Affine transformations
- Outline fonts
- Left-to-right, right-to-left and vertical text layouts
- Antialiasing
- Opacity

1.3 The TinyLine 2D API

These classes define basic 2D graphics concepts like color, transformation, point, path, etc.

Package com.tinyline.tiny2d

Tiny2D	The Tiny2D defines a graphics context that allows an application to draw shapes, images and texts onto a TinyBuffer object.
TinyBuffer	The TinyBuffer class represents a rectangular array of pixels.

TinyColor	The TinyColor class defines colors in the ARGB color space.
TinyFont	The TinyFont class defines a collection of glyphs together with the information necessary to use those glyphs.
TinyGlyph	The TinyGlyph class specifies a glyph representing a unit of rendered content within a font.
TinyHash	The TinyHash class implements a hashtable, which maps keys to values.
TinyMatrix	The TinyMatrix class represents a 2D affine transformation matrix.
TinyNumber	The TinyNumber class wraps a value of the (double) fixed point type in an object.
TinyPaint	The TinyPaint class defines paint servers in the ARGB color space.
TinyPath	The TinyPath class represents a geometric path constructed from straight lines, and quadratic and cubic (Bézier) curves.
TinyPoint	The TinyPoint class specifies a point representing a location in (x, y) coordinate space specified in fixed point precision.
TinyProducer	The TinyProducer interface provides an interface for objects which can produce the image data from the TinyBuffer object (pixels buffer).
TinyRect	The TinyRect class specifies an area in a coordinate space that is enclosed by the TinyRect object's top-left point (xmin, ymin) and down-right point (xmax, ymax) in the coordinate space.
TinyState	A TinyState object encapsulates state information needed for the basic rendering operations.
TinyStop	The TinyStop class implements a gradient stop.
TinyString	The TinyString class represents character strings.
TinyVector	The TinyVector class implements a growable array of objects.

Package com.tinyline.util

TinyInputStream	A data input stream lets an application read primitive TinyLine 2D data types from an input stream.
TinyOutputStream	A data output stream lets an application write primitive TinyLine 2D data types to an output stream.

1.4 Drawing pipeline

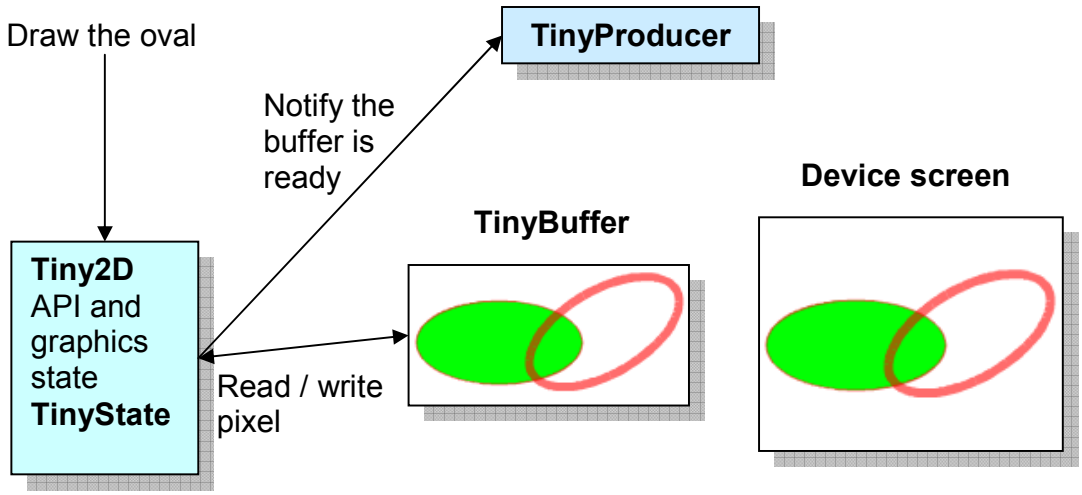
The TinyLine 2D drawing pipeline has four components: `Tiny2D`, `TinyState`, `TinyBuffer` and `TinyProducer`.

The TinyLine 2D rendering process is controlled through the `Tiny2D` object and its state attributes. The state attributes, such as line styles and transformations are applied to graphic objects when they are rendered. The collection of state attributes associated with a `Tiny2D` is referred to as *the graphics state* `TinyState`.

The basic drawing process is the same for any graphics element:

1. Specify the target surface `TinyBuffer`.
2. Specify the appropriate attributes for the graphics by setting the graphics state attributes in the `TinyState` object.
3. Define the shape or text you want to draw.
4. Use the `Tiny2D` object to render the shape, paths or text, by calling one of the `Tiny2D` rendering methods.

During the rasterization, the `Tiny2D` object reads the graphics state attributes `TinyState` and applies them to the drawing process of the graphics primitives. The rasterization target is the `TinyBuffer` object (also named as "canvas") where all is drawn. When the drawing process has completed the `TinyProducer` is notified that new pixels on the `TinyBuffer` "canvas" are ready to be send onto a device screen.



1.5 The painters model

TinyLine 2D uses the painter's model for its imaging. In the painter's model, each successive drawing operation applies a layer of "paint" to an output "canvas" – `TinyBuffer` object. The paint on the pixels buffer (`TinyBuffer` object) can be modified by overlaying more paint through additional drawing operations. This model allows you to construct sophisticated images from a small number of primitives.

When the paint is not completely opaque the result on the pixels buffer is defined by the (mathematical) rules of the *simple alpha blending*.

1.6 Coordinate spaces

TinyLine 2D supports three coordinate spaces: character space, user space and device space. Transformations between coordinate spaces are defined by transformation matrices, which can specify any linear mapping of two-dimensional coordinates.

1.7 Graphics elements

TinyLine 2D supports two basic types of graphics elements that can be rendered onto the `TinyBuffer` object:

- Shapes, which represent some combination of straight line and curves
- Text, which represents some combination of character glyphs

As about raster images, a raster image is presented via the `TinyColor` class as array of values that specify the paint color and opacity (alpha).

Shapes and text can be filled and stroked. Each fill and stroke operation has its own opacity settings; thus, you can fill and/or stroke a shape with a semi-transparently drawn solid color, with different opacity values for the fill and stroke operations.

1.8 Text and fonts

In TinyLine 2D texts are rendered just like shapes. Therefore, coordinate system transformations, painting – all `TinyState` graphics state attributes apply to text in the same way as they apply to shapes or paths.
Additional text attributes include such things like the writing direction (text layout), font specification (`TinyFont`) and painting attributes which describe how exactly to render the characters.
The `TinyFont` class includes the information necessary to map characters to glyphs, to determine the size of glyph areas and to position the glyph area.

1.9 Colors

TinyLine 2D supports the following built-in types of paint that can be used in fill and stroke operations using the ARGB color space:

- Single color

- Gradients (linear and radial)
- Patterns (raster images)

1.10 TinyLine 2D Examples

TinyLine 2D comes with the following examples:

Alias.java	The Alias shows how to draw an oval with anti-aliasing and without it..
Beziers.java	The Beziers is an animated bezier curve. It shows how to change the TinyPath geometry and update only part of the screen..
Colors.java	The Colors shows how to draw with a single ARGB color.
Dash.java	The Dash shows how to draw lines with different line dash patterns.
FillAlpha.java	The FillAlpha shows how to set the opacity of the fill drawing operation.
FillRule.java	The FillRule draws paths with different fillRule values.
GAlpha.java	The GAlpha shows how to set the opacity of all painting operations.
Gradients.java	The Gradients shows how to draw with gradients colors.
HitPath.java	The HitPath shows how to use hitPath function of the Tiny2D. It allows the user to move the path around.
HitText.java	The HitText shows how to use hitPath function of the Tiny2D. It allows the user to move the text around.
LineCap.java	The LineCap shows how to draw lines with different line cap styles.
LineJoin.java	The LineJoin shows how to specify line join styles.
Lines.java	The Lines shows how to draw lines.
Ovals.java	The Ovals shows how to draw ovals.
Paths.java	The Paths draws paths.
Patterns.java	The Patterns shows how to draw with patterns (bitmaps) colors. Also, it shows how to draw on different targets and store and restore the graphics state object.
Polys.java	The Polys draws polygons and polylines.
SmoothBits	The SmoothBits shows how to draw images with anti-aliasing.
TextLines.java	The TextLines shows how to draw text in different directions. Also, it demonstrates how to line-break and draw a paragraph of text.
Transform.java	The Transform shows how to change the current transformation matrix.

2. Fixed Point Numbers

TinyLine 2D uses an efficient fixed-point mathematics. A fixed-point data type is characterized by the word size in bits, the binary point, and whether it is signed or unsigned. The position of the binary point is the means by which fixed-point values are scaled and interpreted. Positive and negative values can also be represented as fixed-point numbers. One bit is used to hold the sign of the number.

TinyLine 2D supports fixed point and double fixed-point numbers. It corresponds to `FIX_BITS` and `DFIX_BITS` binary point. In other words, fixed-point numbers have `FIX_BITS` bits fraction length and double fixed-point numbers have `DFIX_BITS` bits fraction length. Thus, all numbers must be limited in range between `-32,767.9999` to `+32,767.9999`.

The Tiny2D class contains methods for performing basic numeric operations with fixed-point precision numbers:

Name	Description
abs(int)	Returns the absolute value of a `fixed point` value.
max(int, int)	Returns the greater of two `fixed point` values.
min(int, int)	Returns the smaller of two `fixed point` values.
mul(int, int)	Returns a fixed point number whose value is `(a * b)`.
fastDistance(int, int)	Returns the fast approximation for the Euclidean distance between two points, in 2D it is `d = sqrt(x^2 + y^2)`.
div(int, int)	Returns a fixed point number whose value is `(a / b)`.
round(int)	Returns the closest `int` to the argument.
sin(int)	Returns the trigonometric sine of an angle.
cos(int)	Returns the trigonometric cosine of an angle.
tan(int)	Returns the trigonometric tangent of an angle.
atan2(int, int)	Converts rectangular coordinates (`dx`, `dy`) to polar (r, *theta*).

Fixed point arithmetic has advantages and of cause disadvantages as well.

Advantages

1. Fixed point arithmetic is as fast as integer operations.
2. No special hardware required.

Disadvantages

1. Limited range of values.

2. Loss of precision if intermediate result exceeds the maximum value.
3. Programmer must normalize results manually.

3. Coordinate Spaces and Transformations

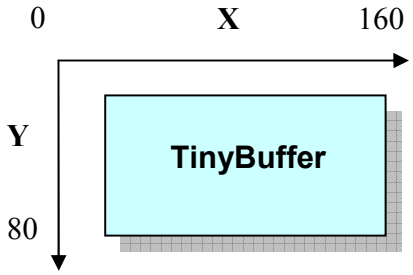
What is a coordinate space? All geometric shapes and positions are defined in terms of pairs of coordinates. A coordinate pair is a pair of numbers x and y that locates a point horizontally and vertically within a two-dimensional coordinate space.

Therefore, a coordinate space is defined by the following properties:
1. The location of the origin (0,0)
2. The orientation of the X and Y axes (directions of axes)
3. The lengths of the units along each axis (length of a step along axes)

TinyLine 2D supports three coordinate spaces: character space, user space and device space. Transformations between coordinate spaces are defined by transformation matrices, which can specify any linear mapping of two-dimensional coordinates.

3.1 Device space

Coordinates in device space specify a particular pixel on the `TinyBuffer` object. In TinyLine 2D device space coordinates are integers. The origin of device space coordinate system is at the upper left corner with the positive direction of the y-axis pointing downward, and the positive direction of the x-axis to the right.



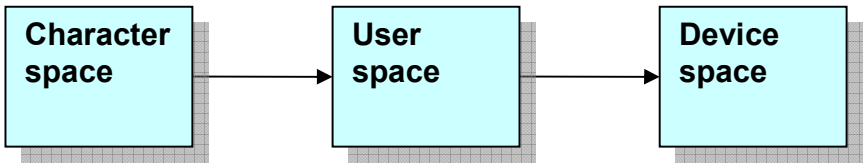
3.2 User space

User space is the logical coordinate system being used in application programs. Almost all methods in the TinyLine 2D API are using coordinates in user space. In TinyLine 2D user space coordinates are fixed point numbers.

3.3 Character space

Characters glyphs (`TinyGlyph`) outlines in the `TinyFont` class are defined in character space. In TinyLine 2D character space coordinates are EM units.

3.4 Relationships among coordinate spaces



The transformation from user space to device space is specified by the current matrix of the `Tiny2D` graphics state.

The transformation from character space to user space is defined by a matrix (cmat) returned by the `charToUserTransform` method of the `Tiny2D` class.

3.5 Transformations

In TinyLine 2D all coordinate transformations, including transformations from character, user to device space, are represented by `TinyMatrix` objects.

The `TinyMatrix` class represents a 2D affine transformation matrix. By modifying a `TinyMatrix`, objects can be scaled, rotated, translated, or transformed. `TinyMatrix` is represented by a transformation matrix written as:

```
[ a    b   0 ]
[ c    d   0 ]
[ tx   ty  1 ]
```

Here tx and ty are single fixed point numbers and `a,b,c,d` are double fixed point numbers.

The transformations could be specified as combinations of those listed below:

- Translation is equivalent to the `TinyMatrix [1 0 0 1 tx ty],` where tx and ty are the distances to translate coordinates in X and Y.

- Scaling is equivalent to the `TinyMatrix [sx 0 0 sy 0 0]`. One unit in the X and Y directions in the new coordinate system equals sx and sy units in the previous coordinate system.
- Rotation about the origin is equivalent to the `TinyMatrix [cos(a) sin(a) -sin(a) cos(a) 0 0]`, which has the effect of rotating the coordinate system axes by angle a.
- A skew transformation along the x-axis is equivalent to the `TinyMatrix [1 0 tan(a) 1 0 0]`, which has the effect of skewing X coordinates by angle a.
- A skew transformation along the y-axis is equivalent to the `TinyMatrix [1 tan(a) 0 1 0 0]`, which has the effect of skewing Y coordinates by angle a.

If several transformations are applied, the order is important. In general, transformations should be done in the following order: translate, rotate, scale.

Coordinate transformations could be expressed as:

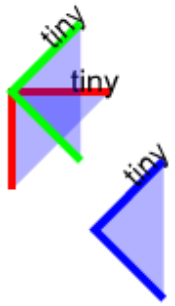
```
                              [  a    b   0 ]
[x' y' 1 ] = [x  y  1]        [  c    d   0 ]
                              [  tx   ty  1 ]
```

By carring out the multiplication:

```
x' = ax + cy + tx;

y' = bx + dy + ty;
```

The transformations can be combined (multiplied) together to produce a single equivalent transformation of the series of transformations.

Example: Transform.java



The example demonstrates how the transformations can be combined together.

To start, we define the first matrix as a translation to the point (50, 50).

```
/* Define the matrix1 as  "translate(50,50)"  */
/* Translation is the TinyMatrix [1 0 0 1 tx ty],
 * where tx and ty are the distances in x and y. */
matrix1.translate(50<<Tiny2D.FIX_BITS,50<<Tiny2D.FIX_BITS);
```

When we draw the red line and text, we set the current transformation matrix to the matrix1:

```
/* Overwrites the Tiny2D state */
tstate.devMat = matrix1;
tstate.fillColor = fillColor;
tstate.strokeColor = redColor;
tstate.strokeWidth = (4 << Tiny2D.FIX_BITS);
/* Draws the path */
t2d.drawPath(path);

/* Overwrites the Tiny2D state */
tstate.fillColor = blackColor;
tstate.strokeColor = TinyColor.NONE;
/* Draws the text */
t2d.drawChars(font, fontSize, engText, 0, engText.length,
                      X, Y, Tiny2D.TEXT_ANCHOR_START);
```

To draw the green line and text we build the second matrix as combination of a translation the point (50, 50) and a rotation to (-45) degrees:

```
/* Define the matrix2 as "translate(50,50) + rotate(-45)" */
matrix2 = new TinyMatrix(matrix1);
m = new TinyMatrix();
/* Rotation is TinyMatrix [cos(a) sin(a) -sin(a) cos(a) 0 0],
 * which has the effect of rotating the coordinate system
 * axes by angle a. */
m.rotate(-(45<<Tiny2D.FIX_BITS),0,0);
/* Concatenates the m  to the matrix2.
 * [matrix2] = [m] x [matrix2]
 */
matrix2.preConcatenate(m);
```

To complicate the things a little when we want to draw the blue line and text we build the third matrix as combination of a translation the point (50, 50) , a rotation to -45 degrees and a translation to the point (-20, 80).

```
/* Define the matrix3 as
 * "translate(50,50) + rotate(-45) + translate(-20,80)"  */
/* Copy the matrix2 to the matrix3 */
matrix3 = new TinyMatrix(matrix2);
m = new TinyMatrix();
/* Translation is the TinyMatrix [1 0 0 1 tx ty],
 * where tx and ty are the distances in x and y. */
m.translate(-(20<<Tiny2D.FIX_BITS),80<<Tiny2D.FIX_BITS);
/* Concatenates the m  to the matrix3.
```



```
         * [matrix3] = [m] x [matrix3]
         */
        matrix3.preConcatenate(m);
```

4. Graphics State - TinyState

The exact effect of the drawing is determined by `TinyState` attributes as the current line thickness (stroke width), the current stroke color, the current fill alpha, etc.

Below is the table of the `TinyState` attributes of the graphics state, sorted alphabetically. For each attribute, the table lists the name of the attribute, the initial (default) value and the description.

Attribute	Initial Value	Description
`antialias`	`true`	If it is true, Tiny2D draws shapes and text with anti-aliasing.
`bg`	`0xffffffff`	The background ARBG color
`capStyle`	`CAP_BUTT`	The stroke cap style specifies the shape to be used at the end of open subpaths when they are stroked. There are three stoke cap styles: `CAP_BUTT`, `CAP_ROUND` and `CAP_SQUARE`.
`dashArray`	`null`	The stroke dash array controls the pattern of dashes and gaps used to stroke paths.
`dashPhase`	`0`	The stroke dash phase specifies the distance into the dash pattern to start the dash.
`devBounds`	`an empty rectangle`	The shape bounds in device space.
`fillAlpha`	`255`	The fill alpha specifies the opacity of the painting operation used to paint the interior the shape.
`fillColor`	`TinyColor(0xff000000)`	The fill color paints the interior of the given shape.
`fillRule`	`FILL_STYLE_EO`	The fill rule indicates the algorithm which is to be used to determine what parts of the canvas are included inside the shape. There are two types of winding rules: `FILL_STYLE_EO` and `FILL_SYLE_WIND`

globalAlpha	255	The global alpha specifies the opacity of all painting operations.
joinStyle	JOIN_MITER	The stroke join style specifies the shape to be used at the corners of paths or basic shapes when they are stroked. There are three line join styles: JOIN_BEVEL, JOIN_MITER and JOIN_ROUND.
matrix	the identity matrix	The transformation matrix that defines the mapping from user space to device space.
miterLimit	4<<FIX_BITS	The stroke miter limit imposes a limit on the ratio of the miter length to the stroke width.
smoothbits	false	If it is true, Tiny2D draws images with anti-aliasing.
strokeAlpha	255	The stroke alpha specifies the opacity of the painting operation used to stroke the shape.
strokeColor	TinyColor.NONE	The stroke color paints along the outline of the given shape.
strokeWidth	1<<FIX_BITS	The width of the stroke.
textDir	TEXT_DIR_LR	The current text direction.

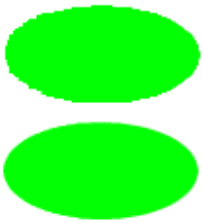
4.1 Antialiasing

Aliasing occurs because vector objects like paths or text outlines have continuous, smooth curves and lines and pixels are discrete and square.
The `Tiny2D` drawing functions include the rasterization process - the process of converting vector data (`TinyPath`) into pixel data (`TinyBuffer`). Since pixels are square and uniformly colored, lines become jagged.

Antialiasing reduces these unwanted jagged borders between colors.

If the `antialias` is true, then `Tiny2D` drawing functions use antialiasing during the rasterization process. Because, antialiasing is computationally expensive, costs CPU, in some cases you might want to turn antialiasing on for some shapes and turn it off for the others.

Example: Alias.java



Before displaying the first oval, the application turns the antialiasing off:

```
/* Turns the antialiasing off */
tstate.antialias   = false;
/* Draws the ovals */
t2d.drawOval(20<<Tiny2D.FIX_BITS,
             40<<Tiny2D.FIX_BITS,
             100<<Tiny2D.FIX_BITS,
             50<<Tiny2D.FIX_BITS);
```

Then we translate the current transformation matrix and turn the antialiasing on before drawing the same oval again:

```
/* Translate the current transformation matrix */
TinyMatrix m = new TinyMatrix();
m.translate(0<<Tiny2D.FIX_BITS, 60<<Tiny2D.FIX_BITS );
tstate.devMat = m;

/* Turns the antialiasing on */
tstate.antialias   = true;
/* Draws the ovals */
t2d.drawOval(20<<Tiny2D.FIX_BITS,
             40<<Tiny2D.FIX_BITS,
             100<<Tiny2D.FIX_BITS,
             50<<Tiny2D.FIX_BITS);
```

4.2 Background color

The `TinyBuffer` object is a surface onto which graphics elements are drawn. We need to fill the `TinyBuffer` object with some initial color (background color) before we can draw anything.

4.3 Line cap style

The line cap style (`capStyle`) specifies the shape to be used at the end of open subpaths when they are stroked. There are three line cap styles: `CAP_BUTT`, `CAP_ROUND` and `CAP_SQUARE`.

The CAP_BUTT line cap style ends unclosed subpaths and dash segments with no added decoration.

The CAP_ROUND line cap style ends unclosed subpaths and dash segments with a round decoration that has a radius equal to half of the width of the pen.

CAP_SQUARE line cap ends unclosed subpaths and dash segments with a square projection that extends beyond the end of the segment to a distance equal to half of the line width.

Example: LineCap.java

In the example, we draw the same line with different line cap styles. The thin white line is drawn only for emphasizing the difference between styles.

First, the line is drawn with the CAP_BUTT line cap style:

```
/* Draws a black line with the CAP_BUTT line cap style */
tstate.strokeWidth = (20<<Tiny2D.FIX_BITS);
tstate.capStyle = Tiny2D.CAP_BUTT;

t2d.drawLine(40<<Tiny2D.FIX_BITS,
             60<<Tiny2D.FIX_BITS,
             140<<Tiny2D.FIX_BITS,
             60<<Tiny2D.FIX_BITS);
```

Then the line is drawn with the CAP_ROUND line cap style:

```
/* Draws a black line with the CAP_ROUND line cap style */
tstate.strokeColor = blackColor;
tstate.strokeWidth = (20<<Tiny2D.FIX_BITS);
tstate.capStyle = Tiny2D.CAP_ROUND;

t2d.drawLine(40<<Tiny2D.FIX_BITS,
             100<<Tiny2D.FIX_BITS,
             140<<Tiny2D.FIX_BITS,
             100<<Tiny2D.FIX_BITS);
```

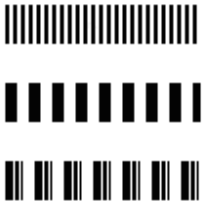

Finally, we draw the line with the `CAP_SQUARE` line cap style:

```
/* Draws a black line with the CAP_SQUARE line cap style */
tstate.strokeColor = blackColor;
tstate.strokeWidth = (20<<Tiny2D.FIX_BITS);
tstate.capStyle = Tiny2D.CAP_SQUARE;

t2d.drawLine(40<<Tiny2D.FIX_BITS,
             140<<Tiny2D.FIX_BITS,
             140<<Tiny2D.FIX_BITS,
             140<<Tiny2D.FIX_BITS);
```

4.4 Line dash pattern

The line dash pattern (`dashArray` and `dashArray`) specifies the pattern of dashes and gaps used to stoke paths.
The `dashArray` defines the lengths of dashes and gaps and the `dashPhase` defines the distance into the dash patterns to start the dash. Both values should be in user space. Dashes wrap around corners and go along curves just as stoked lines and curves do.

Example: Dash.java



In the example we draw the same line but with different line dash patterns. The line drawing code is similar for all cases:

```
/* Draws a black line with the first dash array */
tstate.strokeWidth = (20<<Tiny2D.FIX_BITS);
tstate.capStyle = Tiny2D.CAP_BUTT;
tstate.dashArray = dashArray1;
t2d.drawLine(40<<Tiny2D.FIX_BITS,
             60<<Tiny2D.FIX_BITS,
             140<<Tiny2D.FIX_BITS,
             60<<Tiny2D.FIX_BITS);
```


What is changing is the `dashArray` attribute of the `TinyState` graphics state.

For each line, the corresponding dash array is defined as:

```
int dashArray1[] = { 2<<Tiny2D.FIX_BITS,  2<<Tiny2D.FIX_BITS };
int dashArray2[] = { 6<<Tiny2D.FIX_BITS,  6<<Tiny2D.FIX_BITS };
int dashArray3[] = { 4<<Tiny2D.FIX_BITS,  1<<Tiny2D.FIX_BITS,
                     2<<Tiny2D.FIX_BITS,  1<<Tiny2D.FIX_BITS,
                     1<<Tiny2D.FIX_BITS,  6<<Tiny2D.FIX_BITS };
```

4.5 Shape bounds

The shape bounds in device space `devBounds` specifies what area of the `TinyBuffer` will be updated.

4.6 Fill Alpha

The `fillAlpha` specifies the opacity of the painting operation used to paint the interior of paths and text characters. The valid values should be inside the range `0` (fully transparent) to `255` (fully opaque). The initial value is `255`.

Example: FillAlpha.java

In the example, first we draw the full opaque red rectangle and the full opaque green rectangle over the red one.

```
/* Draws the red rectangle first */
tstate.fillColor = (redColor);
tstate.fillAlpha = 255;
t2d.drawRect(40<<Tiny2D.FIX_BITS,
             40<<Tiny2D.FIX_BITS,
             45<<Tiny2D.FIX_BITS,
```



```
                45<<Tiny2D.FIX_BITS);
/* Draws the green rectangle over the red one */
tstate.fillColor = (greenColor);
t2d.drawRect(20<<Tiny2D.FIX_BITS,
                20<<Tiny2D.FIX_BITS,
                60<<Tiny2D.FIX_BITS,
                60<<Tiny2D.FIX_BITS);
```

Then we shift the current transformation matrix and draw the full opaque red rectangle and the green rectangle with the semi transparent `fillAlpha (127)` over the red one.

```
/* Draws the red rectangle first */
tstate.fillColor = (redColor);
t2d.drawRect(40<<Tiny2D.FIX_BITS,
                40<<Tiny2D.FIX_BITS,
                45<<Tiny2D.FIX_BITS,
                45<<Tiny2D.FIX_BITS);
/* Draws the green rectangle over the red one */
tstate.fillColor = (greenColor);
tstate.fillAlpha = 127;
t2d.drawRect(20<<Tiny2D.FIX_BITS,
                20<<Tiny2D.FIX_BITS,
                60<<Tiny2D.FIX_BITS,
                60<<Tiny2D.FIX_BITS);
```

4.7 Fill Color

The `fillColor` attribute is used to paint the interior of paths and text characters. The initial value is a solid back color `0xff000000` in ARGB format.

4.8 Fill Rule

The `fillRule` attribute indicates the algorithm that is to be used to determine what parts of the canvas are included inside the shape. There are two types of winding rules: `FILL_STYLE_EO` and `FILL_SYLE_WIND`. The initial value is `FILL_STYLE_EO`.

The `FILL_STYLE_EO` fill rule is an even-odd winding rule. A `FILL_STYLE_EO` winding rule means that enclosed regions of the path alternate between interior and exterior areas as traversed from the outside of the path towards a point inside the region.

The `FILL_STYLE_WIND` fill rule is a non-zero winding rule. A `FILL_SYLE_WIND` winding rule means the following. Let us assume an imaginary ray is drawn in any direction from a given point to infinity. If the places where the path intersects the ray are examined, the point is inside of the path if the number of times that the path crosses the ray from left to right does not equal the number of times that the path crosses the ray from right to left.

Example: FillRule.java

In the example, we draw the same path with the same fill color, but with different fill rules.

First, we draw the path with the FILL_STYLE_EO fill rule.

```
/* Sets the fill rule. */
tstate.fillRule = Tiny2D.FILL_STYLE_EO;

/* Draws the path */
t2d.drawPath(path);
```

Then we change the current transformation matrix and draw the path with the `FILL_STYLE_WIND` fill rule.

.

```
/* Sets the fill rule. */
tstate.fillRule = Tiny2D.FILL_STYLE_WIND;

/* Draws the path */
t2d.drawPath(path);
```

4.9 Global Alpha

The `globalAlpha` specifies the opacity of the all painting operations. The value must be in the range `[0-255]`.

The picture below compares a global alpha setting of `127` with the default value of `255`.

Example: GAlpha.java

The example is similar to the FillAlpha.java excepts that we change the alpha for all painting operations when we drawing the second set of rectangles:

```
/* Draws the red rectangle first */
tstate.fillColor = (redColor);
tstate.globalAlpha = 255;
t2d.drawRect(40<<Tiny2D.FIX_BITS,
             40<<Tiny2D.FIX_BITS,
             45<<Tiny2D.FIX_BITS,
             45<<Tiny2D.FIX_BITS);
/* Draws the green rectangle over the red one */
tstate.fillColor = (greenColor);
t2d.drawRect(20<<Tiny2D.FIX_BITS,
             20<<Tiny2D.FIX_BITS,
             60<<Tiny2D.FIX_BITS,
             60<<Tiny2D.FIX_BITS);
```

4.10 Line join style

The line join style `(joinStyle)` specifies the shape is to be used at the corners of stoked paths. There are three line join styles: `JOIN_MITER`, `JOIN_ROUND` and `JOIN_BEVEL`.

The `JOIN_MITER` line join style joins path segments by extending their outside edges until they meet.

The `JOIN_ROUND` line join style joins path segments by rounding off the corner at a radius of half the line width.

The `JOIN_BEVEL` line join style joins path segments by connecting the outer corners of their wide outlines with a straight segment.

Example: LineJoin.java

In the example, the same path is drawn with different line join styles. The thin white line is drawn only for emphasizing the difference between styles.

First, the path is drawn with the `JOIN_MITER` line join style:

```
/* Draws a black path with the JOIN_MITER line join style */
tstate.strokeWidth = (20<<Tiny2D.FIX_BITS);
tstate.joinStyle = Tiny2D.JOIN_MITER;
t2d.drawPath(path, null, null);
```

Then the path is drawn with the `JOIN_ROUND` line join style:

```
/* Draws a black path with the JOIN_ROUND line join style */
tstate.strokeColor = blackColor;
tstate.fillColor   = TinyColor.NONE;
tstate.strokeWidth = (20<<Tiny2D.FIX_BITS);
tstate.joinStyle = Tiny2D.JOIN_ROUND;
t2d.drawPath(path, null, null);
```

Finally, we draw the path with the `JOIN_BEVEL` line join style:

```
/* Draws a black path with the JOIN_BEVEL line join style */
tstate.strokeColor = blackColor;
tstate.fillColor   = TinyColor.NONE;
tstate.strokeWidth = (20<<Tiny2D.FIX_BITS);
tstate.joinStyle = Tiny2D.JOIN_BEVEL;
t2d.drawPath(path, null, null);
```

4.11 Matrix

The `devMat` attribute is the matrix specifying transformation from user space to device space.

4.12 Miter limit

When two line segments meet at a sharp angle and `JOIN_MITER` is specified for the line join style `joinStyle`, it is possible for the miter to extend far beyond the thickness of the line stroking the path.

The `miterLimit` imposes a limit on the ratio of the miter length to the `strokeWidth`. When the limit is exceeded, the join style is converted from `JOIN_MITER` to `JOIN_BEVEL`.

The value of `miterLimit` must be a number greater than or equal to `1<<FIX_BITS`. The initial value is `4<<FIX_BITS`.

4.13 Smooth bits

Aliasing occurs also when drawing with a bitmap color the source image has been transformed (for example scaled). Anti-aliasing reduces unwanted jagged borders between colors.

If the `smoothbits` is true, then `Tiny2D` draws images with anti-aliasing. Because, anti-aliasing is computationally expensive, costs CPU, in some cases you might want to turn anti-aliasing on for some images and turn it off for the others.

Example: SmoothBits.java

Before displaying the first rectangle, the application turns the smoothbits off:

```
/*
 * Draw the first rectangle with the bitmap color without
 * anti-aliasing
 */
tstate.devMat = matrix1;
tstate.fillColor = color;
tstate.strokeColor = TinyColor.NONE;
/* Tiny2D draws images without anti-aliasing. */
tstate.smoothbits = false;
```



```
t2d.drawRect(0 << Tiny2D.FIX_BITS, 0 << Tiny2D.FIX_BITS,
        60 << Tiny2D.FIX_BITS, 40 << Tiny2D.FIX_BITS);
```

Then we translate the current transformation matrix and turn the anti-aliasing on (smoothbits) before drawing the same rectangle again:

```
/*
 * Draw the second rectangle with the bitmap color with
 * anti-aliasing
 */
tstate.devMat = matrix2;
tstate.fillColor = color;
tstate.strokeColor = TinyColor.NONE;
/* Tiny2D draws images with anti-aliasing. */
tstate.smoothbits = true;
t2d.drawRect(0 << Tiny2D.FIX_BITS, 0 << Tiny2D.FIX_BITS,
        60 << Tiny2D.FIX_BITS, 40 << Tiny2D.FIX_BITS);
```

4.14 Stroke Alpha

The `strokeAlpha` attribute specifies the opacity of the painting operation used to paint the border of paths and text characters. The valid values should be inside the range `0` (fully transparent) to `255` (fully opaque). The initial value is `255`.

4.15 Stroke Color

The `strokeColor` attribute is used to paint the border of paths and text characters. The initial value is `TinyColor.NONE` that causes no stroke to be painted.

4.16 Stroke Width

The `strokeWidth` specifies the thickness of the line used to stoke a path or a text and is measured in user space units. A stroke width of `0` causes no stroke to be painted. The initial value is `1<<FIX_BITS`.

4.17 Text layout direction

The characters in certain scripts are written from right-to-left or from top-to-bottom. The `textDir` attribute specifies whether the progression direction for a text should be left-to-right, right-to-left, or top-to-bottom. There are three text direction styles: `TEXT_DIR_LR`, `TEXT_DIR_RL` and `TEXT_DIR_TB`. The initial value is `TEXT_DIR_LR`.

5. Paths and shapes

TinyLine 2D provides a general path (TinyPath), which can be used to create a variety of graphical shapes, and also Tiny2D class provides drawing functions for common basic shapes such as rectangles and ellipses.

These basic shapes drawing functions are convenient for coding and may be used in the same ways as the more general paths.

5.1 Paths

The TinyPath class represents a geometric path constructed from straight lines, and quadratic and cubic (Bézier) curves. Paths are used to represent lines, curves and regions.

A path consists of a series of path segments. Path segments may be straight lines or quadratic and cubic (Bézier) curves. It can contain multiple subpaths.
Path segments operators are: `moveTo`, `lineTo`, `curveTo`, `curveToCubic`, `closePath`. As a result of an applying these segments operators the correspondent segments are created.

Multiple subpaths can be expressed by using a "`moveTo`" segment operator to create a discontinuity in the geometry to move from the end of one subpath to the beginning of the next.

The following function of the Tiny2D class draw the outline of the specified TinyPath object.

Method	Description
`drawPath`	Draws the outline of the specified TinyPath.

Example: Beziers.java

The example animates randomly generated `TinyPath` object.

The `run` function of the `Beziers` class performs animations and it has two parts.
The first part draws the `TinyPath` object on the screen for the first time.

```
/* Draws the scene (first time) if needed. */
if(!drawn)
{
    synchronized (this)
    {
       /* Draws the scene (first time) if needed */
       t2d.invalidate();
       /* Clears the clipRect area before production */
       t2d.clearRect(t2d.getClip());

       /* Overwrites the Tiny2D state */
       tstate.devMat = (mat);
       tstate.fillColor =  fillColor;

       /* Draws the path */
       t2d.drawPath(path, null, null);

       /* Sends the new pixels */
       t2d.sendPixels();
       drawn = true;
    }
}
```

The second part is the animation itself and it calls the `drawDemo` function:

```
/* Calculates and draws the current step. */
drawDemo(imgWidth, imgHeight);
```

The `drawDemo` is the function that produces one animation step. It demonstrates several important things.
First, it shows how to build the `TinyPath` object using the path segments operators:

```
path.moveTo(midx<<Tiny2D.FIX_BITS, midy<<Tiny2D.FIX_BITS);
path.curveToCubic(x1<<Tiny2D.FIX_BITS,y1<<Tiny2D.FIX_BITS,
x2<<Tiny2D.FIX_BITS, y2<<Tiny2D.FIX_BITS, midx<<Tiny2D.FIX_BITS,
midy<<Tiny2D.FIX_BITS);
```

Second, it shows how to update only the damaged area of the `TinyBuffer` object and then send those new generated pixels onto the device screen.

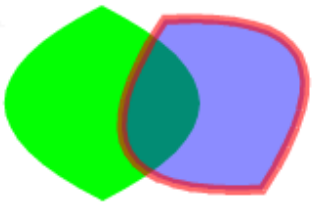
```
/* Clears the dirty area. */
dirtyRect.setEmpty();
bounds = path.getBBox();
TinyRect r = t2d.getDevBounds(mat, strokeWidth, bounds);

/* Adds the old bounds of the shape to the dirty area. */
dirtyRect.union(r);

/* Generates the new pata data. */
```



```
      bounds = path.getBBox();

      /* Adds the new bounds of the shape to the dirty area. */
      dirtyRect.union(t2d.getDevBounds(mat, strokeWidth,
bounds));

      /* Sets the dirty area. */
//    t2d.invalidate(); // only for debug
      t2d.setClip(dirtyRect);

      /* clear the clipRect area before production */
      t2d.clearRect(t2d.getClip());

      /* Repaints the dirty area. */
      t2d.drawPath(path, bounds, null);
      /* Sends the new pixels to the display. */
      t2d.sendPixels();
```

Here the `dirtyRect` rectangle is the clip rectangle in device space.

Example: Paths.java



The example shows how to build `TinyPath` objects by reading from a data stream.

Here is the path data before and after compilation:

```
       /* The SVGT path before compilation (SVGT2Bin)
        * <path id="paths" fill="#45633C"
        * d="M 60 20 Q -40 70 60 120 Q 160 70 60 20 z"/>
        */
       /* The binary path data. */
       byte pathdatabin[] = {6, 0, 0, 0, 121, 30, 0, 5, 0, 31, 96, 0,
       - 116, 0, 60, 0, 60, 0, 53, 0, 1, 24, 0, 120, 0, 20, 0, -96, 0,
       0, 0, 0};
```

Reading the path from the stream:

```
       /* Reads the path data from the stream */
       try
       {
```



```
            ByteArrayInputStream bais =
                new ByteArrayInputStream(pathdatabin);
            TinyInputStream tis = new TinyInputStream(bais);
            path = tis.readTinyPath();
            tis.close();
        }
        catch (IOException ioe)
        {
            System.out.println("Failed to read the stream");
            System.out.println("ioe" + ioe);
            ioe.printStackTrace();
        }
```

In addition, it demonstrates basic things like filling and stroking attributes, changing the current transformation matrix, etc.

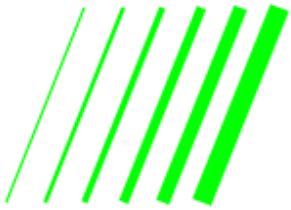
5.2 Basic Shapes

In addition to the paths drawing functions, `Tiny2D` supports the following set of drawing functions for basic shapes:

Method	Description
`drawLine`	Draws a line between the points `(x1, y1)` and `(x2, y2)`
`drawRect`	Draws the outline of the specified rectangle.
`drawRoundRect`	Draws an outlined round-cornered rectangle.
`drawOval`	Draws the outline of an oval.
`drawPolyline`	Draws a sequence of connected lines defined by a vector of points.
`drawPolygon`	Draws a closed polygon defined by vector of points.

Mathematically speaking, each of these basic shapes is equivalent to a path that would construct the same shape.

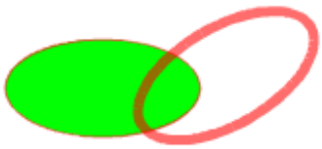
Example: Lines.java



The example draws several lines with growing stoke widths:

```
/* Draws the lines */
tstate.strokeWidth = (1<<Tiny2D.FIX_BITS);
t2d.drawLine(20<<Tiny2D.FIX_BITS,
             150<<Tiny2D.FIX_BITS,
             60<<Tiny2D.FIX_BITS,
             50<<Tiny2D.FIX_BITS);

tstate.strokeWidth = (2<<Tiny2D.FIX_BITS);
t2d.drawLine(40<<Tiny2D.FIX_BITS,
             150<<Tiny2D.FIX_BITS,
             80<<Tiny2D.FIX_BITS,
             50<<Tiny2D.FIX_BITS);
```

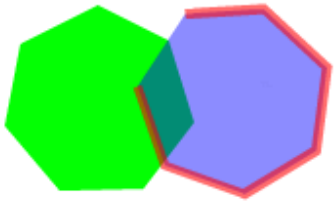
Example: Ovals.java



The example draws the ovals with different fill and stroke colors:

```
/* Overwrites the Tiny2D state */
tstate.devMat    = new TinyMatrix();
tstate.fillRule  = (Tiny2D.FILL_STYLE_EO);
tstate.fillColor = (greenColor);
tstate.strokeWidth = (1<<Tiny2D.FIX_BITS);
/* Draws the first oval */
t2d.drawOval(20<<Tiny2D.FIX_BITS,
             60<<Tiny2D.FIX_BITS,
             100<<Tiny2D.FIX_BITS,
             50<<Tiny2D.FIX_BITS);
```


Example: Polys.java



The example shows how to build vector of points and

```
/* The poins data. */
static int pointsdata[] = {
59,45,95,63,108,105,82,139, 39,140,11,107,19,65 };

/* Fills the points vector */
cnt = pointsdata.length;
i = 0;
points = new TinyVector(1 + cnt / 2);
while (i < cnt)
{
    p = new TinyPoint(pointsdata[i++] << Tiny2D.FIX_BITS,
                      pointsdata[i++] << Tiny2D.FIX_BITS);
    points.addElement(p);
}
```

then draw these points using `Tiny2D` API

```
/* Overwrites the Tiny2D state */
tstate.devMat = (mat1);
tstate.fillColor = (blueColor);
tstate.fillAlpha = (117);
tstate.strokeColor = (redColor);
tstate.strokeAlpha = (150);
tstate.strokeWidth = (5<<Tiny2D.FIX_BITS);

/* Draws the polyline  */
t2d.drawPolyline(points);
```

5.3 Bounds and hit test

Every shape has a bounding box. The bounding box is a rectangle that fully encloses the shape's geometry. Bounding boxes are used to determine whether an object has been selected or "hit" by the user or what area the object occupies on the canvas.

We use `TinyRect` class for bounding boxes. The `TinyRect` class specifies an area in coordinate space that is enclosed by the `TinyRect` object's top-left point (xmin, ymin) and down-right point (xmax, ymax) in coordinate space.

The following example is interesting from many points of view including bounding boxes and coordinate spaces.

Example: HitPath.java

The example shows how to use `hitPath` function for example for moving shape around the canvas.

We will move an oval over the canvas with the help of pointer device (or mouse in the case of the desktop). Make sure that you run this example on devices with a pointer device otherwise, you will not have a fun with it.

Please look at the HitPath.java code.

Because the `hitTest` function in `Tiny2D` class needs a `TinyPath` object as its input, we need to create an oval outline for the later use.

```
/* Create the outline. */
path = Tiny2D.ovalToPath(20<<Tiny2D.FIX_BITS,
                 60<<Tiny2D.FIX_BITS,
                 100<<Tiny2D.FIX_BITS,
                 50<<Tiny2D.FIX_BITS);
```

When the mouse or the pointer has been pressed in the canvas we define if the pixel where the pointer device has been pressed “hits” our path.

```
hit = t2d.hitPath(pressedX, pressedY, path, null);
```

If the path was hit, we want to move it onto the new position. For that, we change the transformation matrix:

```
if(hit)
```



```
        {
           matrix.tx -= (pressedX - x) <<Tiny2D.FIX_BITS;
           matrix.ty -= (pressedY - y) <<Tiny2D.FIX_BITS;
        }
```

6. Text and Fonts

6.1 Introduction

In TinyLine 2D texts are rendered like other graphics elements. Thus, coordinate system transformations, painting – all `TinyState` graphics state attributes apply to text elements in the same way as they apply to shapes such as paths or rectangles.

`Tiny2D` does not perform automatic line breaking or word wrapping. To achieve the effect of multiple lines of text, you need to look at the TextLines.java example that follows at the end of this chapter.

`Tiny2D` provides a low-level interface for drawing texts. The following `Tiny2D` function draws the outline of the character array using the `TinyState` current graphics state. The entire character array is styled using the specified font and font size. The base line of the first character is defined by the position (x,y) and the alignment anchor.

```
public final void drawChars(TinyFont font,
                            int fontSize,
                            char[] ac,
                            int off,
                            int len,
                            int x,
                            int y,
                            int anchor)

      Parameters:
      font - the font object.
      fontSize - the size of the font.
      ac - array of characters.
      off - the initial array offset.
      len - the length of array.
      x - the x-axis coordinate of the current text position
      y - the y-axis coordinate of the current text position
      anchor - the text alignement.
```

6.2 Fonts and Glyphs

As you may noticed, `Tiny2D` uses a font (`TinyFont` object), which has sets of shapes that are associated with characters, to draw text.

The `TinyFont` defines a collection of glyphs together with the information necessary to use those glyphs. The `TinyFont` includes the information necessary to map characters to glyphs, to determine the size of glyph areas and to position the glyph area.

The characteristics and attributes of `TinyFont` correspond closely to the SVG fonts. Various font metrics, such as advance values and baseline locations, and the glyph outlines themselves, are expressed in units that are relative to an abstract square whose height is the intended distance between lines of type in the same type size.

This square is called the EM square and it is the design grid on which the glyph outlines are defined. The value of the units-per-em attribute on the `TinyFont` specifies how many units the EM square is divided into. Common values are, for example, 1000 (Type 1) and 2048 (TrueType, TrueType GX and Open-Type).

The design grid for `TinyFont` fonts, along with the initial coordinate system for the glyphs, has the y-axis pointing upward for consistency with accepted industry practice for many popular font formats.

The `TinyGlyph` class specifies a glyph representing a unit of rendered content within a font.

Each `TinyGlyph` object consists of an identifier (unicode char) along with drawing instructions (`TinyPath`) for rendering that particular glyph. The `horizAdvX` is a metric that is used to describe how the glyph must be placed and managed when rendering text.

How to read TinyFont from TinyLine 2D data stream

In your application you will load TinyFont object using loadFont function defined in PPTiny2DCanvas

```
/* Loads the font */
font = canvas.loadFont("/tarial.bin");
```

where the loadFont() function uses TinyInputStream:

```
TinyFont f = null;
InputStream is = null;
TinyInputStream tis = null;

 . . .

 // Reads and parses the stream
 tis =  new TinyInputStream(is);
 f = tis.readTinyFont();
```


How to write TinyFont to TinyLine 2D data stream

In order to store the TinyFont object to TinyLine 2D data stream you can use SVGT2Bin tool.

SVGT2Bin

1. Reads SVG Tiny 1.1 font from an input stream and parses it into `TinyFont` object.
2. Writes `TinyFont` object into TinyLine 2D data stream.

6.3 Bounds and hit test

As we said before every shape in TinyLine 2D has a bounding box. The same apply to texts as well. The bounding box is a rectangle that fully encloses the shape's geometry. Bounding boxes are used to determine whether or not an object has been selected or "hit" by the user or what area the object occupies on the canvas.
We use `TinyRect` class for bounding boxes.

The following example is interesting from many points of view including bounding boxes and coordinate spaces.

Example: HitText.java

The example shows how to use `hitPath` function for moving the text around the canvas.

We will move the text over the canvas with the help of pointer device (or mouse in the case of the desktop). Make sure that you run this example on devices with a pointer device otherwise, you will not have a fun with it.

Please look at the HitText.java code.

Because the `hitTest` function in `Tiny2D` needs a path as its input, we need to create a text outline for the later use.

```
        /* The text outline */
        path = Tiny2D.charsToPath(font, engText, 0, engText.length,
Tiny2D.TEXT_DIR_LR);
        /* The cmat matrix */
        cmat = Tiny2D.charToUserTransform(path, font, fontSize, X, Y,
Tiny2D.TEXT_ANCHOR_START);
```

In addition, we calculated the `cmat` – transformation matrix from character space to user space.

When the mouse or the pointer has been pressed in the canvas, we define if the pixel where the pointer device has been pressed "hits" our path – the text outline.

```
        hit = t2d.hitPath(pressedX, pressedY, path, cmat);
```

If the path (text outline) was hit, we want to move it onto the new position. For that, we change the transformation matrix:

```
        if(hit)
        {
            matrix.tx -= (pressedX - x) <<Tiny2D.FIX_BITS;
            matrix.ty -= (pressedY - y) <<Tiny2D.FIX_BITS;
        }
```

6.4 Measuring text before drawing

If text measurements are important to your application, it is possible to calculate those using `Tiny2D` functions.

Example: TextLines.java

The example shows how to fit a paragraph of text within a certain width.
We want to fit our text into some shape, for example the bounding box `bbox`.

We go over a long character array, calculating the bounds of accumulated characters and break the current line when we touch the `bbox`.

```
/*
 * How to line-break and draw a paragraph
 */
public void drawTextLines()
{
    /* We want to fit our text into some shape,
     * for example bbox (20.0, 80.0, 120.0, 120.0) */
    TinyRect bbox = new TinyRect( (20<<Tiny2D.FIX_BITS),
                                  (100<<Tiny2D.FIX_BITS),
                                  (120<<Tiny2D.FIX_BITS),
                                  (140<<Tiny2D.FIX_BITS) );

    TinyFont f = arialfont;
    int fontSize = (16<<Tiny2D.FIX_BITS);

    /* We go over a long character array, calculating the
     * bounds of accumulated characters and break the
     * current line when we touch the bbox
     */
    int Y = bbox.ymin;   /* The current line Y */
    int off = 0;         /* The current char pointer */
    int len = 0;         /* The current length of the line */

    while ( (off + len) < longLine.length && Y < bbox.ymax)
    {
         TinyRect cbox = Tiny2D.charsBounds(f, fontSize, longLine,
                         off, off+len, Tiny2D.TEXT_DIR_LR);
         if( (cbox.xmax - cbox.xmin) >  (bbox.xmax - bbox.xmin) )
         {
            /* We need to break the current line */
            len--;
            /* Draw the current line */
            tstate.devMat = new TinyMatrix();
            t2d.drawChars(f, fontSize, longLine, off, off+len,
             bbox.xmin, Y, Tiny2D.TEXT_ANCHOR_START );

            /* Advance one line */
            off += len;
            len = 0;
            Y += (cbox.ymax - cbox.ymin);
         }
         len++;
    }
}
```

In addition, the example shows how to draw text using different text layouts: left-to-right, right-to-left and top-bottom.

```
        /* Draws the vertical CJK text */
              tstate.textDir = Tiny2D.TEXT_DIR_TB;
        tstate.fillColor = color2;
```



```
t2d.drawChars(cjkfont,
              (32<<Tiny2D.FIX_BITS),
               cjkText,0,cjkText.length,
              (120<<Tiny2D.FIX_BITS),
              (80<<Tiny2D.FIX_BITS),
              Tiny2D.TEXT_ANCHOR_START);

/* Draws the horizontal right-to-left text */
tstate.fillColor = (color1);
tstate.textDir = Tiny2D.TEXT_DIR_RL;
t2d.drawChars(hebfont,
              (20<<Tiny2D.FIX_BITS),
               hebText,0,hebText.length,
              (100<<Tiny2D.FIX_BITS),
              (80<<Tiny2D.FIX_BITS),
              Tiny2D.TEXT_ANCHOR_START);
```

7. Colors

7.1 Introduction

In TinyLine 2D paths, texts and basic shapes can be filled (which means painting the interior of the shape) and stoked (which means painting along the outline of the shape) using the current graphics state `TinyState` (including fill color, stroke color, global alpha, fill and stroke alphas, etc).

TinyLine 2D supports the notion of a paint server via the `TinyPaint` class.

With `TinyColor`, you can fill or stroke of shapes and text using:

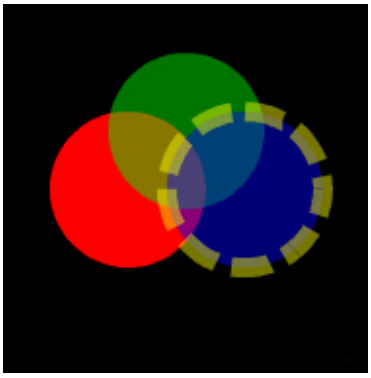
1. Single color in the ARGB color space
2. Linear and radial gradient colors
3. Pattern colors

The `TinyColor` class uses the ARGB color space (Alpha Red Green Blue).

Each pixel value is stored in `0xAARRGGBB` format, where the high-order byte contains the alpha channel and the remaining bytes contain color components for red, green and blue, respectively. The alpha channel specifies the opacity of the pixel, where a value of `0x00` represents a pixel that is fully transparent and a value of `0xFF` represents a fully opaque pixel.

7.2 Single Color

Example: Colors.java



The following code shows how to create opaque and semi opaque single colors in the ARGB color space:

```
/* The solid (full opaque) red color in the ARGB space */
TinyColor redColor    = new TinyColor(0xffff0000);
/* The semi-opaque green color in the ARGB space (alpha is 0x78) */
TinyColor greenColor = new TinyColor(0x7800ff00);
/* The semi-opaque blue color in the ARGB space (alpha is 0x78) */
TinyColor blueColor = new TinyColor(0x780000ff);
/* The semi-opaque yellow color in the ARGB space (alpha is 0x78) */
TinyColor yellowColor = new TinyColor(0x78ffff00);
```

7.3 Gradients

Gradients define continuously smooth color transitions along a vector from one color to the other. TinyLine 2D supports two types of gradients, linear gradients and radial gradients.

7.4 Stops colors

Both linear and radial gradients have color stops. Color stops define the ramp of colors to use on a gradient. The color stop structure is quite simple. It has the stop color ARGB value that indicates what color to use at that gradient stop and the offset value.
The offset value is ranging from 0 to 1 in fixed point numbers which indicates where the gradient stop is placed. For linear gradients, the offset represents a location along the gradient vector. For radial gradients, it represents a percentage distance from the center of the circle.

7.5 Spread method

The spread method points out what happens if the gradient starts or ends inside the bounds of the target region (path, text or basic shape). Possible values are:

`TinyPaint.GRADIENT_PAD` Use the terminal colors of the gradient to fill the remainder of the target region

`TinyPaint.GRADIENT_REFLECT` Reflect the gradient pattern start-to-end, end-to-start, start-to-end, etc.

`TinyPaint.GRADIENT_REPEAT` Repeat the gradient pattern start-to-end, start-to-end, start-to-end, etc.
The default value is `TinyPaint.GRADIENT_PAD`.

7.6 Gradient Units and Gradient transform

The `TinyPaint.unit` attribute defines the coordinate system for gradient vector coordinates.
The gradient units are: `TinyPaint.USER_SPACE_ON_USE` and `TinyPaint.OBJECT_BOUNDING_BOX`.

`TinyPaint.USER_SPACE_ON_USE` defines that the gradient vector coordinates are values in the coordinate system that results from taking the current user coordinate system in place and then applying the gradient transform.

`TinyPaint.OBJECT_BOUNDING_BOX` says that the user coordinate system for the gradient vector coordinates is established using the bounding box of the shape to which the gradient is applied and then applying the gradient transform.

Here the gradient transform is an additional `TinyMatrix` that contains the definitions of an optional additional transformation from the gradient coordinate system onto the target coordinate system. This allows for things such as skewing the gradient. This additional transformation matrix is post-multiplied to any previously defined transformations, including the implicit transformation necessary to convert from object bounding box units to user space.

7.7 Linear Gradient

The linear gradient defines the smooth color transition along a gradient vector defined by starting [x1,y1] and ending [x2,y2] points onto which the gradient stops are mapped. The values of x1, y1, x2, y2 should be in fixed point numbers.

7.8 Radial Gradient

The radial gradient specifies the smooth color transition along a gradient vector defined by the center (x1**,** y1) point of the largest circle and its radius r**.** The radial gradient will be drawn such that the gradient stop at offset 1 is mapped to the perimeter of this largest (i.e., outermost) circle. The values of x1, y1, r should be in fixed point numbers.

Example: Gradients.java

The example shows how to create different solid and opaque gradients and use them for drawing shapes and texts:

```
/* Create the linear gradient color. */
TinyPaint pserver = null;
pserver = new TinyPaint( TinyPaint.FILL_LINEAR_GRADIENT );
/* Add the first stop color */
pserver.addStop(0xffff6600, 13);
/* Add the second stop color */
pserver.addStop(0xffffff66, 242);
/* Create the gradient color ramp */
pserver.createColorRamp();
/* The start point of the linear gradient vector */
pserver.x1 = 50<<8;
pserver.y1 = 50<<8;
/* The end point of the linear gradient vector */
pserver.x2 = 150<<8;
pserver.y2 = 125<<8;
/* The spread method */
pserver.spread = TinyPaint.GRADIENT_PAD;  // default
linColor = new TinyColor( pserver);


/* Create the radial gradient color. */
pserver = new TinyPaint( TinyPaint.FILL_RADIAL_GRADIENT );
/* Add the first stop color */
pserver.addStop(0xffff6600, 13);
/* Add the second stop color */
pserver.addStop(0xffffff66, 128);
/* Add the third stop color */
pserver.addStop(0xffff6600, 256);
/* Create the gradient color ramp */
pserver.createColorRamp();
/* The radial gradient center point */
pserver.x1 = 90<<8;
pserver.y1 = 100<<8;
/* The radial gradient radius */
pserver.r =  50<<8;
```



```
/* The spread method */
pserver.spread = TinyPaint.GRADIENT_REPEAT;
radColor = new TinyColor(pserver);
```

7.9 Patterns

A bitmap image (or sampled image) is an array of pixels (or samples). Each pixel represents a single point in the image. JPEG and PNG graphics files are examples of bitmap images.

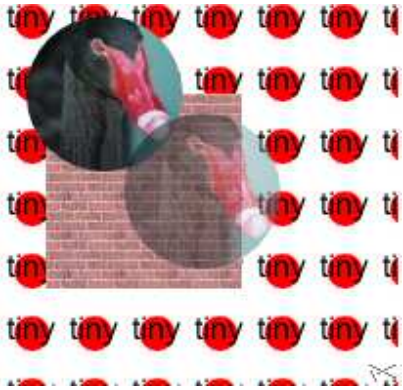
Because of the wide variety of image formats, different Java API and different image formats support on different Java platforms, in order to be portable across all Java flavors, TinyLine 2D does not provide image formats coding and decoding.
We leave these coding and decoding functions outside of the API and operate with the image data instead as it was already decoded from the JPEG or PNG file.

TinyLine 2D uses `TinyBuffer` class for presenting an accessible buffer of image data. In examples, you can find the corresponding helping functions that read JPG or PNG files into `TinyBuffer` objects.

A `TinyBuffer` defines values for pixels occupying a particular rectangular area. The rectangle, known as the `TinyBuffer`'s bounding rectangle, is width, and height values. Each pixel value is stored in `0xAARRGGBB` format, where the high-order byte contains the alpha channel and the remaining bytes contain color components for red, green and blue, respectively. The alpha channel specifies the opacity of the pixel, where a value of `0x00` represents a pixel that is fully transparent and a value of `0xFF` represents a fully opaque pixel.

A pattern color is used to fill or stroke a shape or text using a pre-defined image data (pixels buffer) which can be replicated ("tiled") at fixed intervals to cover the areas to be painted.

Example: Patterns.java



Changing the target surface

Also there is an interesting option to create a `TinyBuffer` object by drawing on it using Tiny2D functions.

We are going to create a "wallpaper" bitmap with the red circle and the word "tiny" and then fill the background with this color. For that we will need to use the following Tiny2D functions:

- Returns the pixel buffer (target surface) for this Tiny2D object.
  `public final TinyBuffer getTarget()`
- Sets the pixel buffer (target surface) for this Tiny2D object.
  `public void setTarget(TinyBuffer pixbuf)`

The following code creates the "wallpaper" bitmap:

```
/* Create the "wallpaper" bitmap */
TinyPaint pserver = null;
bitmap1 = new TinyBuffer();
bitmap1.width = 32;
bitmap1.height = 32;
bitmap1.pixels32 = new int[bitmap1.width * bitmap1.height];

/* Saves the current state and the target */
tmptarget = t2d.getTarget();
copyState(tstate, tmpstate);

/* Set a new target - the "wallpaper" bitmap */
t2d.setTarget(bitmap1);
/* Clear the "wallpaper" bitmap with the solid white color*/
t2d.invalidate();
tstate.bg = 0xffffffff;
t2d.clearRect(tstate.devClip);

/* Set the red fill color */
tstate.fillColor = (redColor);
tstate.strokeWidth = (1 << Tiny2D.FIX_BITS);
/* Draw the oval */
t2d.drawOval(4 << Tiny2D.FIX_BITS, 4 << Tiny2D.FIX_BITS,
             18 << Tiny2D.FIX_BITS, 18 << Tiny2D.FIX_BITS);
/* Set the black fill color */
tstate.fillColor = blackColor;
tstate.strokeColor = TinyColor.NONE;
/* Draw the text */
t2d.drawChars(font, fontSize, engText, 0, engText.length, X, Y,
              Tiny2D.TEXT_ANCHOR_START);

/* The "wallpaper" bitmap is done */
```



```
/* Restores the current state and the target */
copyState(tmpstate, tstate);
t2d.setTarget(tmptarget);
```

Now we can create the corresponding “wallpaper” pattern color:

```
/* Creates the first pattern color - the wallpaper. */
pserver = new TinyPaint(TinyPaint.FILL_PATTERN);
pserver.bitmap = bitmap1;
color1 = new TinyColor(pserver);
```

The next code creates the pattern color for drawing a bird:

```
/* Creates the second pattern color - the bird. */
bitmap2 = canvas.createTinyBuffer(
        new  TinyString("/bird.png".toCharArray()));
pserver = new TinyPaint(TinyPaint.FILL_BITMAP);
pserver.bitmap = bitmap2;
color2 = new TinyColor(pserver);
```

The next code creates the brick wall color:

```
/* Creates the third pattern color - the brick wall. */
bitmap3 = canvas.createTinyBuffer(
        new TinyString("/brick.png".toCharArray()));
pserver = new TinyPaint(TinyPaint.FILL_PATTERN);
pserver.bitmap = bitmap3;
color3 = new TinyColor(pserver);
```

Once we defined the needed pattern colors, we can stoke and fill with a pattern color just the same way we do with a single color or a gradient color.

```
/* Draw the wallpaper */
tstate.fillColor = color1;
t2d.drawRect((tstate.devClip.xmin<<Tiny2D.FIX_BITS),
             (tstate.devClip.ymin<<Tiny2D.FIX_BITS),
             (tstate.devClip.xmax<<Tiny2D.FIX_BITS),
             (tstate.devClip.ymax<<Tiny2D.FIX_BITS) );


/* Draw the brick wall */
tstate.fillColor = color3;
t2d.drawRect((20<<Tiny2D.FIX_BITS),
             (50<<Tiny2D.FIX_BITS),
             (100<<Tiny2D.FIX_BITS),
             (100<<Tiny2D.FIX_BITS) );

/* Draw the first oval with the solid pattern color2 */
tstate.fillColor = color2;
tstate.strokeColor = TinyColor.NONE;
t2d.drawOval(10<<Tiny2D.FIX_BITS,
             10<<Tiny2D.FIX_BITS,
             80<<Tiny2D.FIX_BITS,
             80<<Tiny2D.FIX_BITS);

/* Overwrites the Tiny2D state */
```



```
tstate.devMat   =  matrix2;

/* Draw the second oval with the semi opaque pattern color2 */
tstate.fillColor = color2;
tstate.fillAlpha = 127;
tstate.strokeColor = TinyColor.NONE;
t2d.drawOval(10<<Tiny2D.FIX_BITS,
             10<<Tiny2D.FIX_BITS,
             80<<Tiny2D.FIX_BITS,
             80<<Tiny2D.FIX_BITS);

t2d.sendPixels();
```

8. References

1. Computer Graphics : Principles and Practice Second Edition, James D. Foley, Andries van Dam, Steven K. Feiner, John F. Hughes, Richard L. Phillips, Addison-Wesley, pp. 488-491

2. Adobe Systems Incorporated: PDF Reference: http://partners.adobe.com/asn/acrobat/sdk/public/docs/PDFReference15_v5.pdf

3. TinyLine SVG http://www.tinyline.com/svgt/

4. TinyLine 2D http://www.tinyline.com/2d/

