

Markus Koller

# **Blackspirit Graphics**

**Version 2.0**



# Table of Contents

1	<a href="#">Introduction</a>	1
1.1	<a href="#">Syntax</a>	1
2	<a href="#">Overview</a>	2
2.1	<a href="#">Coordinate System</a>	2
2.2	<a href="#">Example</a>	4
3	<a href="#">Graphics Context</a>	6
3.1	<a href="#">Graphics Listener</a>	6
3.2	<a href="#">Canvas</a>	6
3.2.1	<a href="#">Real-Time Canvas</a>	7
3.2.2	<a href="#">AWT Canvas</a>	8
3.2.3	<a href="#">Canvas Factory</a>	8
3.2.4	<a href="#">Resource Management</a>	9
3.3	<a href="#">Image Graphics Context</a>	9
4	<a href="#">Drawing</a>	10
4.1	<a href="#">View</a>	10
4.2	<a href="#">Transformation</a>	10
4.2.1	<a href="#">Rotation about a specific location</a>	11
4.3	<a href="#">Coloring</a>	11
4.4	<a href="#">Clearing</a>	11
4.5	<a href="#">Image</a>	11
4.5.1	<a href="#">Drawing on Image</a>	12
4.5.2	<a href="#">Image Buffer Manipulation</a>	12
4.6	<a href="#">Text</a>	12
4.7	<a href="#">Primitives (Point, Line, Triangle)</a>	12
4.7.1	<a href="#">Point</a>	12
4.7.2	<a href="#">Line</a>	13
4.7.3	<a href="#">Triangle</a>	13
4.8	<a href="#">Shapes</a>	13
4.8.1	<a href="#">Rectangle</a>	13
4.8.2	<a href="#">Rounded Rectangle</a>	13
4.8.3	<a href="#">Circle</a>	13
4.8.4	<a href="#">Circular Arc</a>	13
4.8.5	<a href="#">Ellipse</a>	13
4.8.6	<a href="#">Ellipsoidal Arc</a>	14
4.9	<a href="#">Drawing Behaviour</a>	14
4.9.1	<a href="#">Drawing Modes</a>	14
4.9.2	<a href="#">Color Masking</a>	14
5	<a href="#">JOGL Implementation</a>	15
6	<a href="#">Setup</a>	16
6.1	<a href="#">Eclipse</a>	16
6.2	<a href="#">Web Start Project</a>	16



TODOs:

- Images for transformations
- Image for text baseline
- Image for texture coordinate system
- Image for mapped ellipse

## 1 Introduction

Blackspirit Graphics is a 2D graphics library for Java, providing a powerful but simple interface. It is suited for real-time applications, but also integrateable into graphical user interfaces. The Blackspirit Graphics core is a set of interfaces allowing for multiple implementations.

The implementation provided is using JOGL (OpenGL bindings for Java) for rendering. Thus it is hardware accelerated and operating system independent.

JOGL Implementation

### 1.1 Syntax

This documentation uses Java syntax to document the library features. To make the documentation easier to read a special syntax is used to simplify the description of overloaded methods. Parameters which are optional to one of the overloaded methods are put in squared brackets. The following example shows a method with two overloads, taking one or two arguments.

```
void example(String first [, String second] );
```

The second example shows a method with three overloads, taking one, two or three arguments.

```
void example(String first [, String second [, String third] ] );
```

## 2 Overview

The core of the Blackspirit Graphics library consists of just a few interfaces, but getting an overview of what the interfaces are doing and how they work together gives a basic understanding.

Overview of the Blackspirit Graphics library (UML)

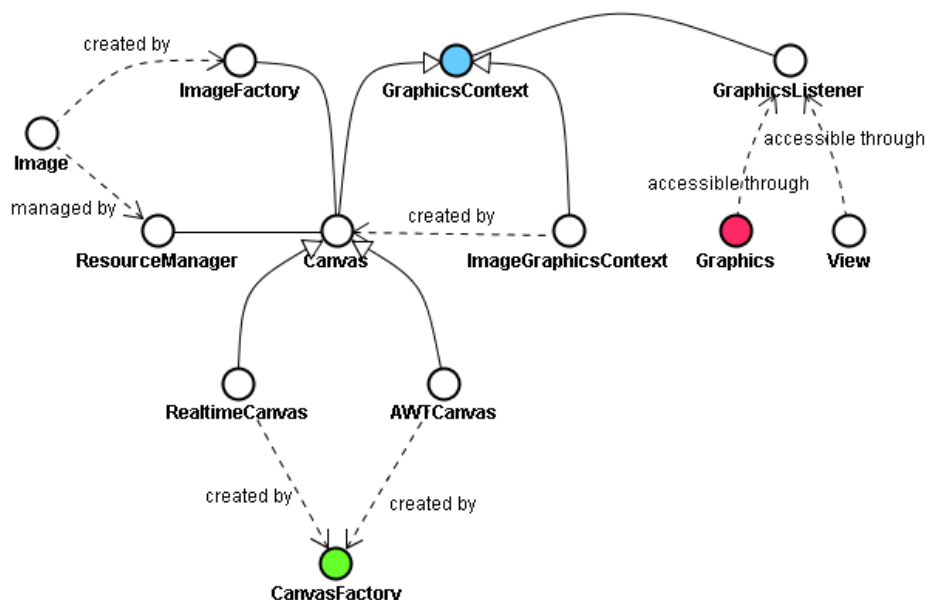


Illustration 1: Overview of the Blackspirit Graphics library in UML

CanvasFactory	The canvas factory is the entry point when using Blackspirit Graphics. It is responsible for creating new canvas' and provides information about available display modes (screen resolution, frame rate).
Canvas	A canvas is a region of the screen where rendering takes place. There are two types of canvas. The real-time canvas is suited for real-time rendering. It can be used fullscreen, which means that the whole screen can be used for rendering, or as a window only containing a region of the screen to render on. The AWT canvas can be integrated in Swing and AWT applications.
ImageGraphicsContext	An image graphics context must be created by the canvas to draw on an image. Each image graphics context is for drawing on a single image.
GraphicsContext	The graphics context is an abstract representation of an area to draw on. This could be the screen (canvas) or an image (image graphics context). The drawing settings (color, transformations, etc.) are being held separately for each graphics context.
GraphicsListener	To do the actual drawing a graphics listener must be implemented which gets triggered when a graphics context gets redrawn. It also notices when a graphics context gets initialised or the size changes.
Graphics	Graphics is the provider of all drawing functionality. It contains methods to draw points, lines, triangles, images and text as well as methods to change the drawing settings like color, transformation and so on.
View	The view describes which region of the coordinate system is visible in a graphics context. Every graphics context has its own view.
ImageFactory	To draw an image it has first to be created using the image factory. An image can be loaded from a URL (network, file system) or it can be empty.

The resource manager allows control over caching and releasing of resources (images, fonts). This is useful to prevent resources from being cached during rendering. When a resource is used for drawing and is not yet cached, it will be cached automatically.

ResourceManager

## 2.1 Coordinate System

The most important thing to now when using Blackspirit Graphics is what coordinate system it is using. In the default coordinate system positive x values are pointing to the right and positive y values are pointing downwards.

!! Image coordinate system

Default Coordinate System

When drawing images, shapes and text you can't directly specify where to draw them. They are always drawn at the coordinate systems origin. In order to draw an image at a specific position or even rotated, the coordinate system itself must be transformed.

Transformations

Every graphics context has it's own coordinate system, so transformations will only affect the graphics context their made on. Transformations will be kept over consecutive calls of the graphics listeners drawing method.



## 2.2 Example

The following example shows the simple usage of Blackspirit Graphics. It creates a canvas to render on and draws an image in the middle of the view. Have a look at the example to get a first impression. It is explained step by step afterwards.

Note: !!! Implement view size for image graphics context using scale

```

1 CanvasFactory canvasFactory =
  new ch.blackspirit.graphics.jogl.CanvasFactory();
2 RealtimeCanvas canvas =
  canvasFactory.createRealtimeCanvasFullscreen();
3 ImageFactory imageFactory = canvas.getImageFactory();
  final Image image = imageFactory.create(...some URL...);
4 canvas.setGraphicsListener(new GraphicsListener() {
5   public void init(View view, Graphics graphics) {
6     view.setSize(800f, 600f);
7   }
8   public void draw(View view, Graphics graphics) {
9     view.setCamera(0f, 0f, 0f);
10    graphics.translate(image.getWidth()/2, image.getHeight()/2);
11    graphics.drawImage(image, image.getWidth(), image.getHeight());
12  }
13  public void resize(Canvas canvas, View view) {}
14 });
15 while(true) {
16   canvas.draw();
17 }

```

Example Explanation

Let's go through it step by step now:

1. A CanvasFactory of the JOGL implementation gets created. The CanvasFactory is the entry point needed to create a new Canvas.
2. A RealtimeCanvas gets created fullscreen, but can later still be changed to be a window. As no display mode is provided it will use the current one.
3. To create an Image an ImageFactory has to be created first. Using the ImageFactory an Image gets loaded from some URL.
4. To do any drawing a GraphicsListener has to be set on the Canvas. In this case it is implemented as an anonymous class. That's why the Image has to be final, otherwise the anonymous class couldn't access it.
5. The initialisation method is only called once for each GraphicsListener that is being set on a Canvas. In this example it sets the views size to 800x600. Drawing is done in its own coordinate system and the view defines how much of it is seen at once.
6. The drawing method is called whenever the canvas gets redrawn. Setting the camera defines which region of the drawing coordinate system is seen in the canvas. Here the camera gets centered at the coordinates (0, 0) with no rotation. Setting the camera is only allowed before any drawing.
7. In the middle of the view the image should be drawn. Images are always drawn at the origin (0, 0) of the drawing coordinate system. To draw an image at certain coordinates, we have to move the coordinate system to make them the new origin. As we want to draw at (-image.getWidth()/2, -image.getHeight()/2) we have to move by (image.getWidth()/2, image.getHeight()/2). This might sound a bit confusing, but will be discussed in more detail later (see 4.2).
8. If the size of a canvas changes the resize method gets called, giving the opportunity to change the view. Here we're not doing anything
9. This is the main drawing loop, every time the draw method gets called, a redraw of the canvas is triggered.

## 3 Graphics Context

A graphics context is an abstract representation of an area to draw on. There are implementations to draw on screen or on images. The different implementations will get covered in detail later.

The drawing settings (color, transformations, etc.) are held separately for each graphics context, so changing the settings will only affect the graphics context the changes are made on.

Every graphics context allows to manually trigger a redraw.

```
void draw();
```

Trigger Drawing

To do the actual drawing a graphics listener can be set. Whenever the canvas is redrawn or resized the graphics listener will be noticed. Only one graphics listener at a time can be set.

```
void setGraphicsListener(GraphicsListener listener);
```

```
GraphicsListener getGraphicsListener();
```

Setting a GraphicsListener

Each graphics context has a region to draw on. The size of this region (in pixels) can be retrieved from the graphics context.

```
int getWidth();
```

```
int getHeight();
```

### 3.1 Graphics Listener

All the drawing is done by a graphics listener. Every time a graphics context is redrawn the drawing method gets called. The graphics object gives access to all drawing operations and settings.

```
void draw(View view, Graphics graphics);
```

Drawing Method

All settings that were changed on the view or on the graphics object during a call of the drawing method will still have the same values in the next call (including transformations).



What the content is of the buffer being drawn on when this method is called is unspecified.



The view must be changed before any drawing. There is no guarantee what will happen if it gets changed afterwards.



Before a graphics listeners drawing method is called the first time the initialisation method will be called, allowing to do any configuration that will not change.

Initialisation Method

```
void init(View view, Graphics graphics);
```

When the size of canvas changes, the graphics listeners resize method is called. This allows for reaction on size changes to change the view for example.

Resize Method

```
void resize(Canvas canvas, View view);
```

### 3.2 Canvas

The Canvas is a graphics context used to render on the screen. There are two types of Canvas, one to do real-time rendering (RealtimeCanvas) and one to integrate in graphical user interfaces (AWTCanvas).

Image Factory	In addition to the standard graphics context functionality, the canvas gives access to an image factory. It is used to create empty images or load existing ones.
	<code>ImageFactory getImageFactory();</code>
Resource Manager	Resources get cached automatically the first time they get used. To handle caching and releasing of resources yourself, the canvas gives access to the resource manager. It allows to manually cache and release resources.
	<code>ResourceManager getResourceManager();</code>
Image Graphics Context	To draw on images an image graphics context can be created. Multiple graphics contexts for the same image can be created, allowing to draw different things on the same image. All resources are shared with the canvas.
	<code>ImageGraphicsContext createImageGraphicsContext(Image image);</code>
Vertical Synchronisation	Some graphics cards show flicker when drawing is not synchronized with the displays refresh cycle. To prevent this you can enable vertical synchronisation. When enabled, drawing will block until the screen got refreshed. Then it is up to you, not to waist processor time waiting for synchronisation. As turning vertical synchronisation on or off is not always supported you have to call <code>getVSync()</code> to see the result. If possible vertical synchronisation is enabled by default.
	<code>void setVSync(boolean enabled);</code>  <code>boolean getVSync();</code>

### 3.2.1 Real-Time Canvas

The real-time canvas is suited for real-time rendering. It can not be integrated into a graphical user interface. It either creates it's own window or is displayed fullscreen.

Fullscreen	When used fullscreen, the whole screen can be used for rendering. It can either use the current display mode or switch it. Available display modes can be retrieved from the canvas factory (see 3.2.3).
	<code>void setFullscreen();</code>  <code>void setFullscreen(DisplayMode displayMode)</code>  <code>boolean isFullscreen();</code>
Window	As a window it only contains a region of the screen. The size of this region can be specified and is the actual size in pixels that will be available to render on. The displayed window has a title which can be changed.
	<code>void setWindow(int width, int height);</code>  <code>void setWindowTitle(String title);</code>  <code>boolean isWindow();</code>
Window State Changes	To have control over the window state a window listener can be added. Window listeners inform about state changes like activation, deactivation, closing and others
	<code>void addWindowListener(WindowListener listener);</code>  <code>boolean removeWindowListener(WindowListener listener);</code>  <code>List&lt;WindowListener&gt; getWindowListeners();</code>



Even when fullscreen, there is still a window, so window listeners work for every real-time canvas.

### 3.2.2 AWT Canvas

The AWT canvas can be integrated into Swing and AWT applications. Independent from the type of application it is being integrated into (AWT or Swing) it can be light- or heavyweight. Swing uses lightweight components, so to properly integrate you will have to use the AWT canvas lightweight.

In order to make the integration of the AWT canvas as easy as possible, the AWT Component where the rendering takes place can be accessed directly. Just add it to an AWT or Swing user interface as usual.

User Interface Integration

```
Component getComponent();
```

An AWT canvas knows whether it is light- or heavyweight. This can not be changed after creation.

Lightweight / Heavyweight

```
boolean isLightweight();
```

### 3.2.3 Canvas Creation

AWT and real-time canvas' are being created by the canvas factory. This is the entry point when using Blackspirit Graphics.

Real-time canvas' are displayed immediately after creation. Therefore it must already be decided on creation whether it should be displayed fullscreen or as a window.

Creating a Real-Time Canvas

There are two methods allowing for creation of a real-time canvas fullscreen. One uses the current display mode and the other you specifically give one. Available display modes can be retrieved from the canvas factory.

```
RealtimeCanvas createRealtimeCanvasFullscreen();
```

```
RealtimeCanvas createRealtimeCanvasFullscreen(DisplayMode mode);
```

When creation a real-time canvas as a window, all you have to provide is the size it should have.

```
RealtimeCanvas createRealtimeCanvasWindow(int width, int height);
```

A newly created AWT canvas is not displayed, therefore you have integrate the canvas into a user interface (AWT or Swing) and make it visible. The size of the screen region to render on changes depending on the user interface you integrated the canvas in. Whether an AWT Canvas should be light- or heavyweight must be specified at creation and can not be changed afterwards.

Creating an AWT Canvas

```
AWTCanvas createAWTCanvas(boolean lightweight);
```

In addition to the canvas creation methods the canvas factory provides information about the systems display modes. The currently active display mode and all available modes can be retrieved.

Display Modes

```
DisplayMode getDisplayMode();
```

```
List<DisplayMode> getDisplayModes();
```

Choosing a Blackspirit Graphics implementation is as easy as instantiating the corresponding canvas factory, everything else you work with are interfaces.

Choosing a Canvas Factory

In the case of the provided JOGL implementation the following code snippet would create the canvas factory.

```
CanvasFactory canvasFactory =
    new ch.blackspirit.graphics.jogl.CanvasFactory();
```

### 3.2.4 Resource Management

In programs using graphics, resource management is an important thing to keep in mind. Currently there are two types of resources, images and AWT fonts. Resources are being stored in system and graphics card memory raising two issues.

#### Resource Caching

First of all transferring this resources to system or graphics card memory is time consuming. When done during rendering it can result in a long waiting period for the first frame (where a load screen should be displayed) or a slowdown. To prevent this, the resource manager allows for manual resource caching.

```
boolean cacheImage(Image image);
```

```
boolean cacheFont(Font font);
```



The methods return whether caching the resource was successful. There is no guarantee that caching resources is possible. Any guarantees given are subject to the implementations.

#### Resource Release

The second issue is that graphics card memory is limited and therefore only a limited number of resources can be stored. If there are too many resources cached they will have to be swapped to system memory or even to disk which results in slowdown. To prevent this, all unused resources should be released. There are methods to release a single resource or all that are cached.

```
boolean releaseImage(Image image);
```

```
boolean releaseImages();
```

```
boolean releaseFont(Font font);
```

```
boolean releaseFonts();
```



As with caching resources there is no guarantee that releasing resources is always possible. However, there is a guarantee that resources get released as soon as possible. The methods return whether releasing the resources was successful. Any other guarantees given are subject to the implementations.

#### Cached Resources

The resource manager keeps track of all cached resources, so you don't have to do this yourself.

```
List<Image> getCachedImages();
```

```
List<Font> getCachedFonts();
```



Blackspirit Graphics can automatically cache resources the first time they're used, but as it has no knowledge about when resources get used they are not released automatically.

### 3.2.5 Image Creation

#### Creating an image

Images get created using the image factory. There are two methods to create images. The first loads an existing image. If forcing alpha is not set to true, there will not necessarily be an alpha channel used internally if not needed by the image. The second method creates an empty image (black), for example to draw on. Whether the image should contain an alpha channel must be specified.

```
Image createImage(URL url, boolean forceAlpha);
```

```
Image createImage(int width, int height, boolean alpha);
```

Buffered images hold the image data in memory. It is easy to access and manipulate without using drawing methods. To load a buffered image from an existing image it can be specified whether an alpha channel should be forced (like loading a unbuffered image). The buffer type can also be explicitly specified. Available buffer types for a Blackspirit Graphics implementation are listed in a BufferTypes class.

Creating a buffered image

```
Image createBufferedImage(URL url, boolean forceAlpha);

Image createBufferedImage(URL url, BufferType bufferType);
```

When creating an empty (black) buffered image it must be specified whether an alpha channel should be available or not, which results in an appropriate buffer type being chosen. A buffer type can also be explicitly specified.

```
Image createBufferedImage(int width, int height, boolean alpha);

Image createBufferedImage(int width, int height,
    BufferType bufferType);
```

### 3.3 Image Graphics Context

To render on images there is the image graphics context. A new image graphics context for an image can be created on the canvas.

```
ImageGraphicsContext getImageGraphicsContext(Image image);
```

Besides the standard graphics context functionality there are some image specific methods.

Like on the canvas a graphics listener can be set on the image graphics context which gets called back when the image graphics contexts draw method is called.

```
public void draw();
```

The view can be used when drawing on images as well. Instead of the defined view being mapped to a screen region it is being mapped to the complete image.

When the graphics listeners drawing method gets called, the buffer is in an unspecified state. If the intention is to draw on the original content of the image, it simply must be drawn first using the image drawing methods. The content of the image will not change until the drawing method has finished.

Drawing on existing image

## 4 Drawing

All drawing is done in the graphics listeners drawing method. It provides access to the view and the graphics object, which contains all the drawing methods.

The following chapters will talk about the view and all the drawing methods in detail.

### 4.1 View

Only a region of the drawing coordinate system is shown on the screen at once. Which region this is can be configured using the view.

The camera defines which drawing coordinates should be shown in the center of the screen. In addition an angle can be set, by which the camera is rotated clockwise about its center.

Camera

```
void setCamera(float x, float y, float angle);
```

```
float getCameraX();  
float getCameraY();  
float getCameraAngle();
```

Size

How much should be visible on the screen at once can be configured by setting the width and height in drawing coordinates.

```
void setSize(float width, float height);  
float getWidth();  
float getHeight();
```

## 4.2 Transformation

Transformation is used to draw on different locations without changing the information given to the drawing methods. Understanding transformation is very important as you will use it all the time, unless you only want to draw at the coordinate systems origin.

Every transformation gets applied at the origin of the current drawing coordinate system. The easiest way of dealing with transformations is to imagine the new coordinate system after each transformation to see the result of another transformation (see 4.2.1).

Translation

The most used transformation is the translation which moves the coordinate system along the x and y axis.

```
void translate(float x, float y);
```

image before after

Rotation

Rotation is applied clockwise around the drawing coordinate system origin. Degrees are used as unit.

```
void rotate(float angle);
```

image before after

Scaling

Scaling resizes the drawing coordinate system with 1 as normal size. For example setting both scales to 2 would cause everything to be drawn twice as big.

```
void scale(float x, float y);
```

image before after

Clearing Transformations

All transformations can be cleared which restores the original drawing coordinate system.

```
void clearTransformations();
```

All the transformations by themselves are pretty straightforward to use but combining them can be tricky. Therefore the next chapter will show and explain a quite common example.

### 4.2.1 Rotation about a specific location

Let's assume we want to draw an image of size (width, height) at coordinates (left, top). The image should be rotated by 90° around its center.

```
Graphics g;  
g.translate(-(left + width/2), -(top + height/2));  
g.rotate(90);  
g.translate(width/2, height/2);  
g.drawImage(image, width, height);
```

Explaining the above transformations with words is very complicated and hard to understand. I hope the following images will make the explication more clear. The images show the old (black) and new coordinate system (red) and where the image would be drawn after this transformation.

Explanation of the transformations

#### 1. Translation picture:

The first translation moves the coordinate system to the images center, because that's where we want to rotate about. The values are negative because we want the images center to become the new origin. Or in other words, we move the images center to the origin.

#### Rotation picture:

We now rotates the coordinate system around it's new origin which is the images center.

#### 2. Translation picture

When drawing an image its top-left corner is drawn at the coordinate system origin. Therefore we have to move the intended location of the image to the coordinate system origin.

## 4.3 Coloring

The color used for drawing operations can be changed. It is used for all drawing operations like drawing points, lines, triangles, images, fonts except for drawing lines with a color per points specified.

```
void setColor(Color4f color);
```

```
void getColor(Color4f color);
```

The base coloring can be used to affect all drawings even if they use different colors. Mathematically speaking the final drawing color (dc) used is the color (c) multiplied with the base color (bc) each component separately.

Base color

$$dc_r = bc_r \cdot c_r$$

$$dc_g = bc_g \cdot c_g$$

$$dc_b = bc_b \cdot c_b$$

$$dc_a = bc_a \cdot c_a$$

If, for example, the base color is set to only blue, just the blue component of everything drawn is visible.

```
void setBaseColor(Color4f color);
```

```
void getBaseColor(Color4f color);
```

## 4.4 Clearing

The whole drawing area can be efficiently cleared with a single color. It is often useful to start a rendering cycle with clearing the drawing area as this is not done automatically.

```
void clear();
```

The color to clear the drawing area with can be specified.

```
void setClearColor(Color4f color);
```

```
void getClearColor(Color4f color);
```

## 4.5 Image

Drawing images is one of the main functionalities of Blackspirit Graphics.

Drawing images

As the size of the image in pixels does not necessarily have a relation to the view size, the width and height the image should be drawn with must be explicitly specified. Optionally the image can be flipped vertically, horizontally or both.

```
void drawImage(Image image, float width, float height [, Flip flip]);
```

Drawing sub images

Often only a part of an image needs to be drawn (a sub image). In addition to the parameters explained above the sub images top left corner and size must be specified in pixels.

```
void drawImage(Image image, float width, float height,
    int subImageX, int subImageY,
    int subImageWidth, int subImageHeight [, Flip flip]);
```

### 4.5.1 Drawing on Image

Drawing on an image is pretty much the same as drawing on the canvas but using an image graphics context. For more information about drawing on images see 3.3.

### 4.5.2 Image Buffer Manipulation

Buffered images provide direct access to the image content without using drawing methods. This is for example useful when rendering a video. Whether an image should have a buffer or not must be specified at creation time (see 3.2.5).

Updating image cache and buffer

When modifying an images buffer the cache has to be updated before the new content gets used. Updating only a region of the cache is also possible.

```
void updateCache( [int width, int height]);
```

Drawing on buffered images using the image graphics context is possible, but the drawings will not be visible in the buffer until it is updated.

```
void updateBuffer( [int width, int height]);
```

Accessing the image buffer

Before accessing the buffer, it should be checked if one is present and what type it actually is. If speed is not important, the buffer type provides methods to access single pixels without having to know about the internal data representation of the buffer, but it is slow.

```
boolean isBuffered();
```

```
BufferType getBufferType();
```

The fastest way of accessing the buffer is getting the buffer object and casting it to the actual object (i.e. byte[]). Therefore the actual buffer object has to be known and buffer type specific handling code needs to be written.

```
Object getBuffer();
```

## 4.6 Text

Text drawing is used in most applications, therefore the interface must be very easy to use. Simply tell it what text to draw. The origin of the current drawing coordinate system is used as the most left point on the baseline (like writing on a line).

```
void drawText(String text);
```

To know where exactly the text should be placed, the text bounds can be retrieved without drawing it. Transformations don't get taken into account.

```
void getTextBounds(String text, Rectangle2D bounds);
```

There is always one font set to draw text. If the font is not already cached, it will get cached automatically when first used. The default font is the systems standard sans serif font.

```
void setFont(Font font);
```

```
Font getFont();
```

## 4.7 Primitives

In 2D Graphics the most used drawing operations, besides drawing images and text, is drawing points and lines. Though they might not be the best from a performance point of view, they're pretty useful.

Triangles are mostly used in 3D but they are just as useful in 2D. Every object can be represented by a polygon (up to some detail level) and polygons can always be represented as triangles. So triangles can be used to draw any type of object.

### 4.7.1 Point

The size of a point is always one pixel and can not be changed. If a circle needs to be drawn, shapes should be used.

```
void drawPoint(float x, float y);
```

### 4.7.2 Line

The easiest way to draw a line is using the x and y coordinates of the two endpoints. As color the currently set drawing color will be used.

```
void drawLine(float x1, float y1, float x2, float y2);
```

The line interface gives the possibility to give each endpoint of a line a different color (which will be used together with the base color). It contains the two endpoints of the line and optionally the color to be used for each endpoint. If no color is set for a point, the currently set drawing color will be used for that point.

```
void drawLine(Line line, boolean useColor);
```

Using the line interface multiple lines can be drawn at once. A big performance advantage can be expected from this.

```
void drawLines(Line[] lines, boolean useColor);
```

The line interface is defined as following:

```
public interface Line {  
    public Vector2f getPoint(int index);  
    public Color4f getColor(int index);  
}
```

The line width is always one pixel. Shapes or triangles should be used to draw thicker lines.

Line Width

### 4.7.3 Triangle

Filling a triangle

The easiest way to fill an area is using triangles. A single triangle can simply be drawn giving the x and y coordinates of its corners.

```
void fillTriangle(float x1, float y1, float x2, float y2,
                 float x3, float y3);
```

Filling a triangle using the triangle interface

The triangle interface gives the possibility to give each corner of a triangle a texture coordinate and a different color (which will be used together with the base color). If no color is set for a point, the currently set drawing color will be used for that point. Texturing can of course only be done when an image is supplied to use as texture. If one of the corners of a triangle does not have texture coordinates the triangle will not be textured.

```
void fillTriangle(Triangle triangle, boolean useColor
                 [, Image texture]);
```

Using the triangle interface multiple triangles can be drawn at once. A big performance advantage can be expected from this.

```
void fillTriangles(Triangle[] triangles, boolean useColor
                  [, Image texture]);
```

The triangle interface is defined as following:

```
public interface Triangle {
    public Vector2f getPoint(int index);
    public Vector2f getTextureCoordinate(int index);
    public Color4f getColor(int index);
}
```

Texture coordinates

Texture coordinates are stored as pixel coordinates (0, 0) being the top left and (width, height) being the bottom right pixel of the image.

!!! Image coordinate system !!!

Drawing triangle outlines

Triangle outlines can also be drawn. The easiest way is again to simply give the x and y coordinates of the three corners. The triangle interface can also be used to draw the outlines.

```
void drawTriangle(float x1, float y1, float x2, float y2,
                 float x3, float y3);

void drawTriangle(Triangle triangle, boolean useColor);

void drawTriangles(Triangle[] triangles, boolean useColor);
```

## 4.8 Shapes

Shapes are not one of the primitives directly supported by Blackspirit Graphics, they are utility classes helping to draw shapes based on triangles and lines. Shapes can also contain cut-outs as they consist of a set of triangles covering the shapes area and multiple set of lines describing the shapes outline.

The shape interface looks as following:

```
public interface Shape extends Area, Lines {}

public interface Area {
    public Triangle[] getTriangles();
    public void fillArea(Graphics graphics, boolean useColors,
                        boolean texturing);

    public void setTexture(Image texture);
    public Image getTexture();
}
```

```
public interface Lines {
    public Line[] getLines();
    public void drawLines(Graphics graphics, boolean useColors);
}
```

The shape interface provides drawing methods which make the appropriate calls on the the graphics interface.

There exists one implementation so far which is called "SimpleShape". It simple creates a shape by passing the triangle and line array to the constructor.

Shape Implementation

A shapes outline can be drawn with optional coloring per point.

Drawing shapes

```
void drawLines(Graphics graphics, boolean useColors);
```

A shapes area can be drawn with optional coloring and texturing. This uses triangles as implementation and coloring and texturing work as they do for normal triangles. A texture must be set on the shape, otherwise no texturing is done.

```
void drawArea(Graphics graphics, boolean useColors,
    boolean texturing);
```

```
void setTexture(Image texture);
```

```
Image getTexture();
```

### 4.8.1 Creating Common Shapes

Common shapes can be easily created using the static methods of the shape factory. The center of the created shapes is always drawn at the drawing coordinate systems origin (0, 0).

Shape Factory

The rectangle is clearly one of the most commonly used shapes. Everything needed to create a rectangle are its width and height.

Rectangle

```
Shape createRectangle(float width, float height);
```

Very close to the normal rectangle is the rounded rectangle. It has round corner which looks very nice in some occasions. In addition to width and height, the corner radius must be specified. The corners will be made up of circle quarters with the specified radius. By default 9 points get used to create the outline and triangles of each corner. For large shapes that might not be sufficient, therefore the number of points to use per corner can be specified.

Rounded Rectangle

```
Shape createRoundedRectangle(float width, float height,
    float cornerRadius [, int pointsPerCorner]);
```

A circle can simply be created by specifying the radius. By default 40 points get used to create the shapes outline and triangles. For large circles that might not be sufficient, therefore the number of points to use can be specified.

Circle

```
Shape createCircle(float radius [, int points]);
```

A circular arc is a piece of a circle (imagine a piece of a round cake). The radius of the circle as well as the start and end angle (in degrees) have to be specified. Where 0° means horizontally to the right. The following image shows an example.

Circular Arc

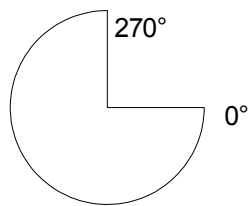


Illustration 2: A circular arc with  $0^\circ$  as start and  $270^\circ$  as end angle.

By default at most 40 points will be used to create the shapes outline and triangles, depending on how many degrees are specified. For large shapes that might not be sufficient, therefore the number of points to use can be specified.

```
Shape createCircularArc(float radius,
    float startAngle, float endAngle [, int points]);
```

#### Ellipse

An ellipse is simply spoken a flattened circle. The width and height have to be specified for creation. By default 40 points get used to create the shapes outline and triangles. For large ellipses that might not be sufficient, therefore the number of points to use can be specified.

```
Shape createEllipse(float width, float height [, int points]);
```

#### Ellipsoidal Arc

An ellipsoidal arc is a piece of an ellipse (imagine a piece of a cake in form of an ellipse). The width and height of the ellipse as well as the start and end angle (in degrees) have to be specified. Where  $0^\circ$  means horizontally to the right. The following image shows an example.

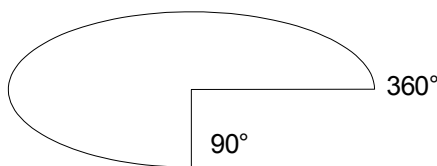


Illustration 3: An ellipsoidal arc with  $90^\circ$  as start and  $360^\circ$  as end angle.

By default at most 40 points will be used to create the shapes outline and triangles, depending on how many degrees are specified. For large shapes that might not be sufficient, therefore the number of points to use can be specified.

```
Shape createEllipsoidalArc(float width, float height,
    float startAngle, float endAngle [, int points]);
```

## 4.8.2 Creating Shapes Manually

#### Shape Creator

Common shapes can be created using the shape factory, but for everything special the shape creator is a great help.

The shape creator currently contains only a single static method creating shapes from a list of points representing the outline and multiple lists of points representing cut-outs which must be inside the outline. The points contained in one of this lists describe the lines belonging to a polygon. Two consecutive points build a line and the last point gets connected to the first to close the polygon.

```
Shape create(List<Vector2f> outline, List<List<Vector2f>> cutouts);
```

Actually the shape factory is using the shape creator as well. All the shape factory is doing, is preparing the list containing the outline points.

### 4.8.3 Mapping Textures

To map a texture on a shape the texture mapper can be used. It currently has a single static method to map a texture without deformation. The texture can be moved, rotated and scaled. The x and y parameters specify what coordinates the textures center should be mapped to. It is then rotated around and scaled from the center.

Texture Mapper

```
void mapTexture(Area area, Image texture, float x, float y,
               float rotationAngle, float scale);
```

The following image shows an image containing a color gradient being mapped an ellipse.

!! image mapped ellipse

## 4.9 Drawing Behaviour

In addition to the previously explained drawing operations, Blackspirit Graphics gives the possibility to change the drawing behaviour allowing for various interesting effects.

### 4.9.1 Drawing Modes

When doing any drawing, what's drawn gets somehow combined with what already in the buffer on a per color component basis (red, green, blue, alpha). This behaviour can be changed.

```
void setDrawingMode(DrawingMode drawingMode);
```

```
DrawingMode getDrawingMode();
```

The following table contains the 5 possible values for drawing mode.

Alpha Blend	This is the default drawing mode. The drawn pixel color gets combined with the buffers pixel color based on the drawn pixels alpha value. The lower the drawn pixels alpha value, the more of the buffers color and less of the drawn pixel color gets use, making it look transparent.
Add	Each drawn and buffer pixel color get added on a per component basis. The value range for a color component is from 0 to 1. If the new value is outside this range it gets truncated.
Subtract	Each drawn pixel color gets subtracted from the buffer pixel color on a per component basis. The value range for a color component is from 0 to 1. If the new value is outside this range it gets truncated.
Multiply	Each drawn and buffer pixel color get multiplied on a per component basis. The value range for a color component is from 0 to 1. If the new value is outside this range it gets truncated.
Overwrite	The buffer pixel simply gets overridden by the drawn pixel.

### 4.9.2 Color Masking

Normally drawing takes place on every color component (red, green, blue, alpha). This can be changed by setting the color mask. By default everything is set to true, but by setting one color component to false that color component of each pixel will just be left as it is, no matter what drawing operation takes place.

```
void setColorMask(boolean red, boolean green, boolean blue,  
                 boolean alpha);  
  
void setRedMask(boolean red);  
boolean getRedMask();  
  
void setGreenMask(boolean green);  
boolean getGreenMask();  
  
void setBlueMask(boolean blue);  
boolean getBlueMask();  
  
void setAlphaMask(boolean alpha);  
boolean getAlphaMask();
```

## **5 JOGL Implementation**

## **6 Setup**

### **6.1 Eclipse**

### **6.2 Web Start**

## Illustration Index

Illustration 1: Overview of the Blackspirit Graphics library in UML.....	2
Illustration 2: A circular arc with 0° as start and 270° as end angle.....	16
Illustration 3: An ellipsoidal arc with 90° as start and 360° as end angle.....	16