いまさら聞けない組み込みGUI!!

Go To 新(ニュー) GUI 世界(ワールド)

トランジスタ技術
特別小冊子(非売品)

麻生勝之／宮崎仁［共著］

CQ出版社

アクセルのAIソリューション

組み込み機器向けグラフィックスLSI「AG9シリーズ」

AG903

アクセルのLSI「AG903」は、Arm® Cortex®-A5 CPUとOpenVG™ 1.1対応の高性能グラフィックスエンジン、1Gbit のDRAM、画像処理エンジン等を搭載した、産業機器向けSoCです。

■ ailia SDK for RTOS

ailia SDKはアクセルおよびその子会社ax株式会社が開発する、推論に特化したディープラーニング推論エンジンです。同 for RTOSは、組み込み機器向けに学習データを含めたソースコードを出力することが可能です。

- RTOSやNon-OSで使用できる推論SDK
- C言語(C99)のソースコードを出力するため幅広い環境で動作

■ AG903＋ailia

一般的にPC・サーバーに比べ、組み込みSoCは非力とされます。しかし、このアクセル製品の組み合わせでは、RTOSを搭載するような組み込みシステムにおいても、高速、高精度な判定を行うAI機能を実現しています。また、アクセルの子会社 ax 株式会社では、組み込み機器向けに最適な学習データ作成から請け負え、さらなる高いパフォーマンスを搭載させることが可能です。

- 応用例　物体検出（色、大きさ等）、性別・年齢推定、文字認識

■ AG903搭載標準ボード「AG903-C」

株式会社コスモが販売するAG903-Cは、AG903（AX51903G）を搭載した標準プラットフォームのボードです。SoCが持つ性能を生かしたUI、カメラモニタリング、画像処理、AIといった様々な機能を必要とするお客様の機器にそのまま組み込んでご使用頂ける製品です。ailiaのAIお試しソフトも用意しており、組み込み機器におけるAIを、簡単に導入可能です。

AG903-C 仕様一覧

搭載 SoC	AG903（AX51903G）
ビデオ I/F	アナログ入力　最大4ch デジタル1ch（入出力排他仕様、DVI I/F） LVDS（Single 2ch / Dual 1ch）
オーディオ I/F	入出力各 1ch
ペリフェラル I/F	Ethernet、USB2.0、UART、RS485
カード I/F	CF、SD
デバック I/F	JTAG

AG903（AX51903G）仕様概要

CPU コア	Arm® Cortex®-A5
描画エンジン	OpenVG™ 1.1 対応
内蔵 VRAM	1Gbit（128M バイト）
ビデオ入力	デジタル×1、アナログ×4
ビデオ出力	デジタル、LVDS（Dual / Single）
最大解像度	1,920×1,200
画像コーデック	JPEG（圧縮伸長）、H.264（伸長）
その他	画像前処理エンジン、外部CPUからの制御

お問い合わせはこちら

COSMO 株式会社コスモ

E-mail:cosmo-sales@cosmo.co.jp

AG903 搭載標準ボード「AG903-C」CEB-AX51903GA

ORコードよりホームページにアクセスできます。▶

AXELL CORPORATION

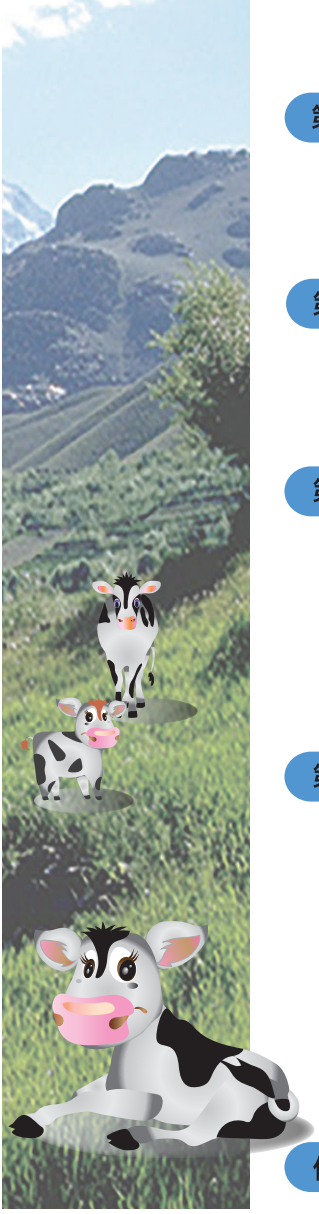
株式会社アクセル
〒101-8973 東京都千代田区外神田4-14-1 秋葉原 UDX 南ウイング10階
TEL 03-5298-1670　FAX 03-5298-1671　EMAIL：sales@axell.co.jp

詳しくはこちらへ
https://www.axell.co.jp/

目 次
[CONTENTS]

いまさら聞けない組み込みGUI!!

麻生勝之／宮崎仁［共著］



第１章 写真館

ホームコントローラ・トップ画面 …… 2
ホームコントローラ・カメラ画面 …… 4
ホームコントローラの画面遷移例 …… 5

第２章 組み込みGUI概論 …… 6

2-1 組み込みGUIとは …… 6
2-2 CPUとユーザインタフェースの接続 …… 8
2-3 組み込みGUIの開発環境 …… 10

第３章 AG903を用いたGUIの実現方法の解説 …… 12

3-1 画面の構成 …… 12
3-2 画面の更新 …… 15
3-3 画面の遷移 …… 17
3-4 ボタン操作とアニメーション表示 …… 19
3-5 テロップ表示 …… 20

第４章 AG903内蔵描画機能の操作方法 …… 23

4-1 OpenVGとは …… 23
4-2 AG903内蔵OpenVGアクセラレータの使い方 …… 24
4-3 OpenVGの座標系 …… 30
4-4 OpenVGによる描画領域の指定 …… 32
4-5 OpenVGによる描画例 …… 34
4-6 OpenVGによる背景，画像の事前作成 …… 54
4-7 OpenVGによるアニメーションの例 …… 57
4-8 OpenVGによる文字の描画 …… 58
4-9 OpenVGによる色変換と合成 …… 61

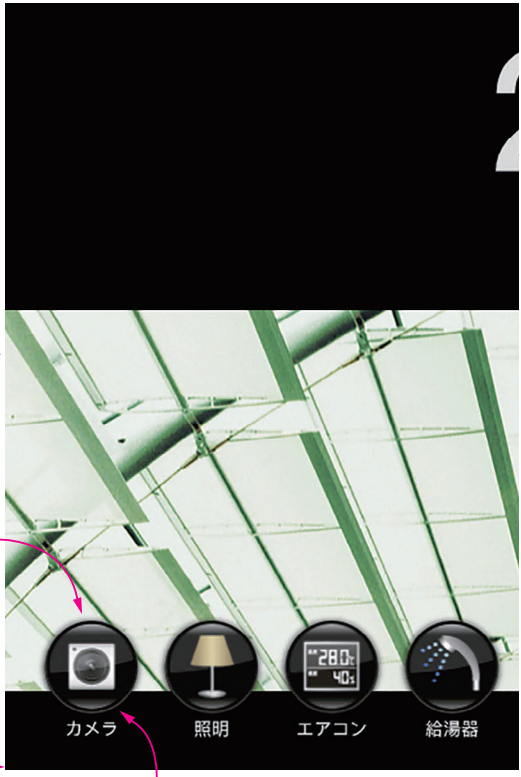
付 録 AG903の概要 …… 63

第1章 写真館 ホームコントローラ・トップ画面

ドアホンカメラや監視カメラ，照明，エアコン，給湯器，テレビなど，住宅機器や家電機器を一か所で操作できるホームコントローラの事例．液晶とタッチパネル一体の組み込み機器で，直感的で分かりやすい表示と操作性，高いデザイン性が求められる．

スライドショーの画像を VRAM に格納後，VGImage オブジェクトにする．vgCreateImage と vgImageSubData を利用する場合はメモリコピーを伴うが，AG903 では拡張 API の vgCreateDirectImageEXT を利用することで，メモリコピー無しに VGImage オブジェクトを生成できる．

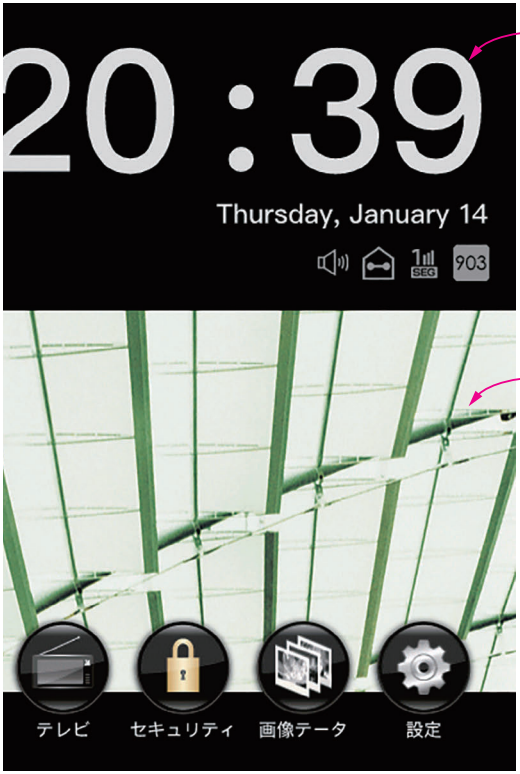
ボタンはアルファ付き画像．ボタンの外部は α=0%（透過），ボタン周囲はアンチエイリアス処理され α=100%→0% に変化．背景とは VG_BLEND_SRC_OVER モードで合成する．



描画先原点

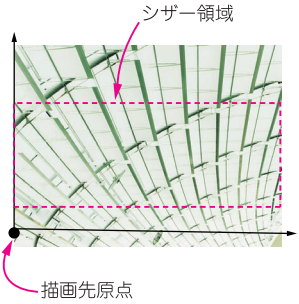
VRAM に格納した 11 フレームの全ボタン画像を親の VGImage オブジェクトとし，vgChildImage でフレーム分割して子の VGImage オブジェクトを生成する．ボタンのタッチにより，子の VGImage オブジェクトを逐次選択してアニメーション表示する．

ボタンのタッチパネル操作はアニメーションにより視認性を高め，操作後の画面遷移は次の画面が中央から現れる凝った作りとなっている．OpenVG を用いることで多彩なGUIの実現が可能だ．どのようなAPIが利用されているかp.2～5に示した．



フォントはダイナコムウェア社の金剛黒体フォントを使用し，DigiType API により ttf ファイルから OpenVG のパスを生成．時刻は文字単位で，日付とボタン名称は文字列として vgSetGlyphToPath で VGFont オブジェクトに登録．それぞれ vgDrawGlyph で描画した．

スライドショーは vgTranslate で 0.1px 単位でゆっくり下から上に描画領域を移動させる．0.1px 単位で期待通りの描画を行うため VG_IMAGE_QUALITY_BETTER を指定しておく必要がある．
画面上の表示領域は固定されているため，シザー領域（描画先原点の位置とサイズは固定）を指定して，その領域のみ描画されるようにする．



ホームコントローラ・カメラ画面

カメラ名の表示枠は OpenVG のユーティリティ・ライブラリ（VGU）の vguRoundRect を使用. ビデオキャプチャ画像に重ねるため, α=0.5(半透明)の黒で塗りつぶした.

カメラ画面の背景は凹型の VGPath オブジェクトを作成し，上に行くほど黒くなるリニアグラデーションの VGPaint オブジェクトで塗りつぶす. ビデオキャプチャの表示は AG903 のウィンドウ機能による合成で表示するため，ビデオキャプチャの領域はα=0(透過)にして vgClear で塗りつぶす. 画面中央上側に選択中のカメラ名をビデオキャプチャ画像に重ねて表示させるため，描画サーフェス用ウィンドウ(Win#0)の下にビデオキャプチャ用ウィンドウ(Win#1)を重ねる.

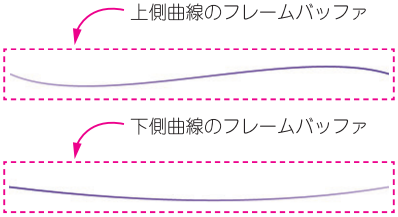
ビデオキャプチャ領域をα=0 で塗りつぶす

Win#0 : 描画サーフェス

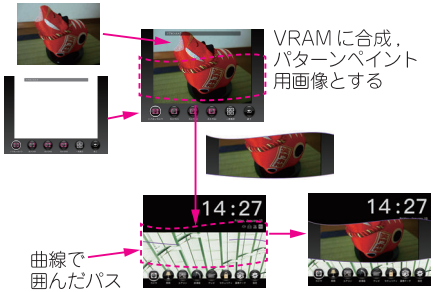
Win#1 : ビデオキャプチャ

ホームコントローラの画面遷移例

画面左右から紫色の曲線を徐々に出現させ，交差させる．2 つの曲線は予め VRAM 内に作成しておき，AG903 のウィンドウ機能で表示する位置と範囲を逐次設定する．各曲線は，EGL で上側曲線と下側曲線用の描画サーフェスを作成し，背景を α=0（透過）で塗りつぶした上，VGPath オブジェクト（ベジェ曲線）で作成する．



2 つの曲線が交差した後，その間から次の画面を出現させる．次の画面がカメラ画面の場合，ビデオキャプチャ画像は表示機能で合成しているため，描画とビデオキャプチャそれぞれのフレームバッファから表示画面と同じイメージで VRAM に複製する．それをパターンペイントのパターン画像として，上下の曲線で囲んだ VGPath オブジェクトを塗りつぶす．その VGPath オブジェクトは，現在の描画サーフェスに重ね書きする．なお，描画サーフェスは現在画面と次画面用に 2 つ作成する．



第2章 組み込み GUI 概論

本章では，組み込み GUI とはどういうものか，組み込み GUI を実現するために必要なハードウェアの構成例，組み込み GUI 開発のための環境やツールなどについて紹介します．

2-1 組み込み GUI とは

グラフィックディスプレイとポインティングデバイスを用いて，直感的にコンピュータを操作するという GUI（Graphical User Interface）の概念は，1973 年に Xerox 社のパロアルト研究所で生まれました．ただし，当時はハードウェアの性能が低く，システムのコストも高くなるため，すぐに普及したわけではありません．

1980 年代にはワークステーションや Mac，1990 年代には Windows PC などで GUI が採用され，広く普及しました．さらに，それらの汎用のコンピュータ製品と比べて CPU 性能や表示デバイス，入力デバイスなどのリソースが限られている組み込み機器においても，GUI は次第に普及してきました．

組み込み GUI は，たとえば銀行の ATM，駅の自動券売機など公共的な場所で，不特定多数のユーザが利用する組み込み機器で使われています．FA や制御機器などの産業用途，カーナビや情報家電などの民生用途の組み込み機器でも広く使われています（**写真 2-1**）．

コンピュータでは，GUI が用いられる前は，テキストやグラフィックを表示可能な CRT などのディスプレイと，文字入力用のキーボードが使われていました．GUI の入力デバイスとしては，ディスプレイ上の X-Y 座標を直接指し示せるポインティングデバイスが必要です．ディジタイザ，マウス，タッチパネルなどのデバイスが登場し，その中で小型，低価格でキーボードとの持ち替え使用が容易にできるマウスが普及しました．

一方，従来の組み込み機器ではメカニカルな表示用ランプや操作用スイッチを個別に備えたシステムが多く，ディスプレイやキーボードを備えたものは少数でした．

コンピュータと組み込み機器

コンピュータは，CPU，メモリ，入出力デバイスなどの汎用のハードウェアをもち，ソフトウェアによって任意の機能を実行する装置です．ソフトウェアはユーザ自身が開発したり，市販のアプリを購入して利用します．

組み込み機器は，内部に組み込まれたハードウェアとソフトウェアで，固有の機能を実行する機器です．たとえば，ユーザが購入後に任意のアプリを入れられるスマートフォンはコンピュータ，購入時の機能を変更できない携帯電話機は組み込み機器です．

写真 2-1　組み込み GUI が使われている例

(a) 銀行の ATM 	銀行の ATM, 駅の自動券売機では, 1 台の装置で複数の機能を実行するために, 階層型の画面とタッチパネルによるメニューシステムを採用している. GUI ならメカニカルな操作ボタンは不要で, 各階層で必要なメッセージやボタンをプログラマブルに表示して, 不特定多数のユーザが容易に直感的に操作できる.
(b) 工場の制御盤 	工場の制御機器では, システムの動作状態の表示や機器の操作を行う制御盤が使われている. 以前は, 多数のランプ類やスイッチ類を並べた制御盤を, システムごとに特注していた. グラフィックディスプレイとタッチパネルを用いて自由に機能を変更できるプログラマブル表示器が普及し, 簡単に制御盤を実現できるようになった.
(c) カーナビ 	カーナビや情報家電など, 民生用途の組み込み機器でも GUI が広く用いられている. カーナビではグラフィックディスプレイの利点を最大限に活用して, メニュー表示と高精細の地図表示の切り替えや, 地図上にさまざまな情報のオーバレイ表示を行い, ドライバに負担をかけない直感的なタッチ操作を可能にしている.

組み込み機器では, GUI を実現するために, グラフィックディスプレイとポインティングデバイスを一体化したタッチパネルを用いるものが多くなっています. コンピュータでも, タブレット PC やスマートフォンのように小型化, モバイル化が進むと, 装置と一体化したタッチパネルが主流となってきます.

2-2 CPU とユーザインターフェースの接続

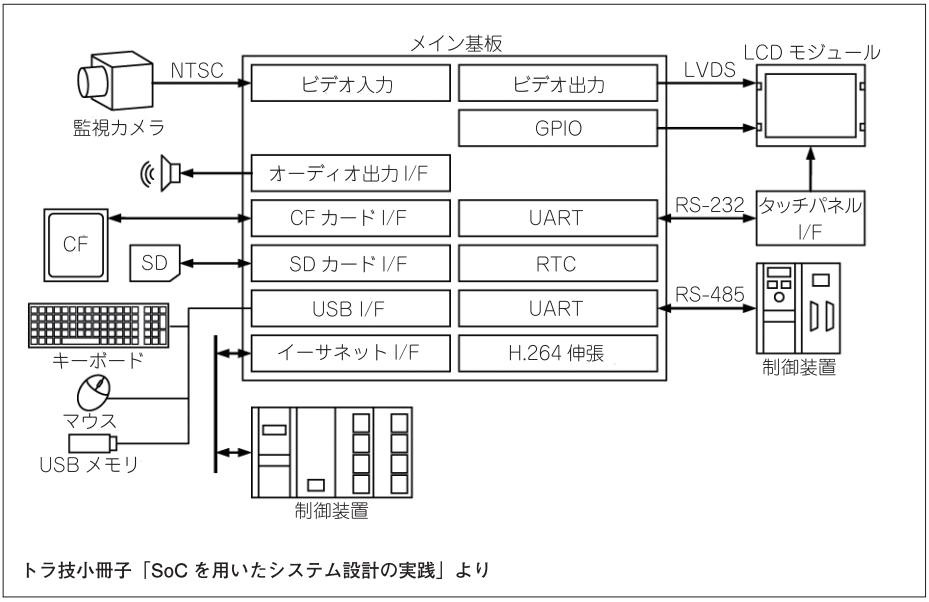
組み込み GUI で用いられる UI(User Interface)用デバイスとして，最も一般的なものは，タッチ入力対応の LCD（液晶ディスプレイ）モジュールや，有機 EL ディスプレイモジュールでしょう．産業用途，民生用途に応じて，さまざまな製品が選択できます．

表示部分のグラフィックディスプレイは，表示専用のディスプレイモジュールと共通性が高く，外付け型ディスプレイは HDMI，DVI，DP(Display Port)，アナログ RGB などの画像インタフェースを備えています．また，機器内蔵型ディスプレイは CMOS RGB（シングルエンド），LVDS（差動）などの画像インタフェースが一般的で，携帯電話向けの小型高精細モジュールでは MIPI DSI を用いるものもあります．

入力部分のタッチパネルは，RS-232 や USB を用いてインターフェースするものが一般的です．外付け型ディスプレイの場合も，画像 I/F ケーブルとは別にタッチパネル I/F ケーブルを接続することが必要です．

図 2-1 に，内蔵型の LCD モジュールを使用したプログラマブル表示器のシステム構成例を示します．ここでは，メイン基板の LVDS ポートを使用して LCD モジュールに画像を表示し，タッチパネルの座標情報はメイン基板の UART(RS-232)で取り込んでいます．なお，LCD モジュール用電源およびバックライト電源もメイン基板から供給し，GPIO 出力を用いてバックライト制御を行っています．

図 2-1　プログラマブル表示器に使用するメイン基板の構成例



トラ技小冊子「SoC を用いたシステム設計の実践」より

図 2-2　RS-232 インターフェース回路

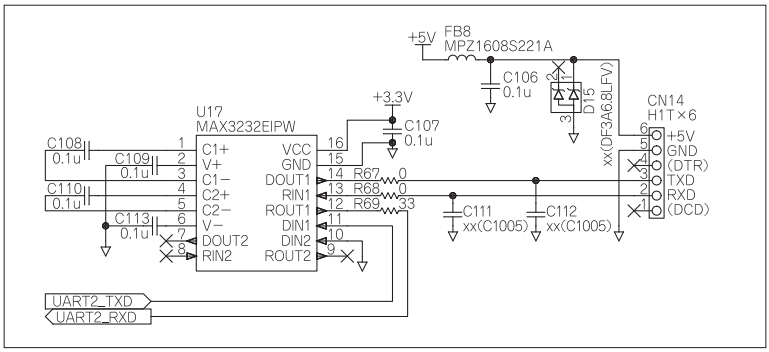
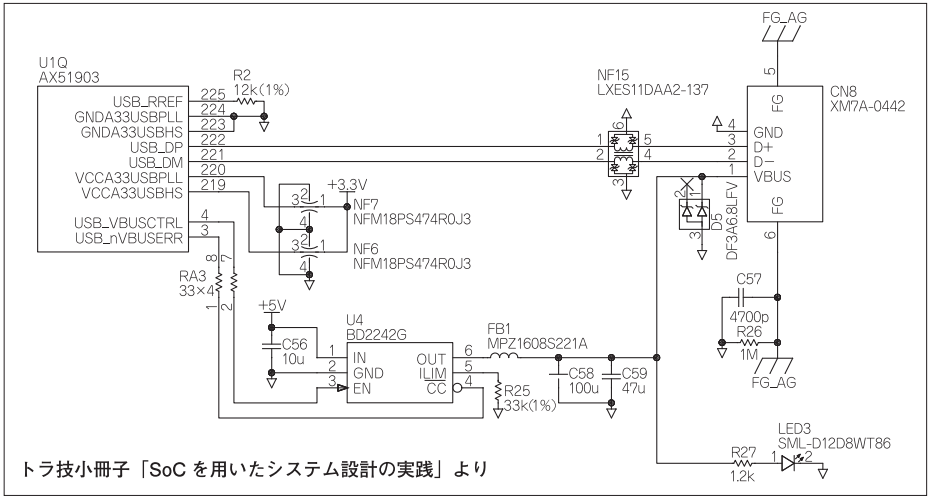


図 2-2 に RS-232 インターフェース回路の例を示します. ここでは，メイン基板に搭載した CPU の UART ポートが 3.3V ロジックレベルなので，MAX3232E（TI）によって RS-232 レベルに変換しています. タッチパネル用の電源供給も必要になります. タッチパネルのインターフェースは 5V CMOS レベルも多く, なんらかのレベル変換は必要になるでしょう.

RS-232 はソフトウェアが簡単にできることが大きな利点です. 組み込みの分野では，USB は有償の OS やドライバソフトが必要になります. 一方，USB にはキーボードやマウス，USB メモリを接続できるメリットがあります. ハブを介して複数のデバイス接続も可能です.

図 2-3 に USB ホストインターフェース回路の例を示します．USB ホストコントローラ内蔵 CPU であれば, 少ない部品点数で回路を実現できます.

図 2-3　USB ホストインターフェース回路



トラ技小冊子「SoC を用いたシステム設計の実践」より

2-3 組み込み GUI の開発環境

組み込み GUI を実現するには，CPU，表示デバイス，入力デバイスなどのハードウェアや組み込み機器の機能に合わせて，GUI のソフトウェア開発を行うことが必要です．

PC やスマートフォンは，GUI のために十分なハードウェアをもち，Windows，macOS，Linux，Android など GUI 対応の OS で使用することが前提です．汎用的な GUI の開発環境やライブラリが用意されており，実現は容易です．

組み込み機器の場合は，機器による違いが大きく，GUI の開発環境はそれほど整っているとは言えません．CPU やメモリ容量などハードウェアのリソースが限られているため，OS なしのシステムが多数です．OS を搭載する場合には，リアルタイム応答性を重視した RTOS や組み込み向けに大幅に軽量化した組み込み Linux が一般的です．

一般に，組み込み向けの開発環境は Arm などの CPU コアメーカやチップメーカ，RTOS や組み込み Linux などの OS ベンダ，各種のツールベンダから提供されています．その中で GUI 向けのものは，それほど多いわけではありません．また，GUI ツールは比較的専門性が高く，供給しているベンダも限られています．

組み込み GUI を重視して開発環境を選ぶ場合，プラットフォーム横断的に使えることを目指した統合的な製品，ポイントツールで一般的な開発環境に対応した製品，CPU や OS に特化した製品などがあります．プラットフォーム横断的な製品は，広い範囲の組み込み機器や PC などの情報機器と設計や UI を共通化できる利点がありますが，一般に高性能のハードウェアを要求されます．リソースの制限が厳しくなるほど，利用できる開発環境や設計方法も制約を受けます．

プラットフォームに依存しない標準規格を利用することによって，異なる環境で GUI 設計を共通化する方法もあります．

たとえば，オープンに利用できる標準的なグラフィックス API として，2 次元グラフィックスの OpenVG，3 次元グラフィックスの OpenGL があります．さまざまな分野の組み込み機器や，PC，スマートフォンなどで利用できますし，PC 上で開発したソースプログラムを組み込み機器に移植することも可能です．

本書の第 4 章では，OpenVG の概要と実際の描画事例について解説します．

インタラクティブなアニメーションなどの動画では，以前は Adobe 社の Flash がデファクトスタンダードとして広く使われていましたが，セキュリティの問題からサポートが終了したことから，他の規格への移行が必要となっています．

現在では，より公的な規格である HTML5 を利用することが多くなっています．ただし，いずれも PC としては軽量な処理ですが，組み込み機器でハードウェア非依存で動画を扱うためには，高性能のハードウェアが必要となります．

●組み込み GUI の特徴と注意事項

組み込み GUI は，その利用分野や対象とするユーザ，使用可能なハードウェアなどによって，スマートフォンのように高画質の表示が望まれるものから，シンプルなボタンなどを単純に配置したものまで，千差万別です．それに合わせて実現方法や開発環境を検討する必要があります．

限られたハードウェアのリソースの範囲内で，可能な限り高度な表示を実現しようとすれば，そのハードウェアに特化してぎりぎりまで最適化を進める必要があります．ただし，開発に大きな手間がかかるだけでなく，ソフトウェアの機能追加やハードウェアの変更に対応できなくなる危険があります．

PC やスマートフォンと違って，組み込み機器ではグラフィック表示はもちろん，文字フォントも自前で用意しなければならない場合が多くなります．文字フォントは既成のものを利用できますが，比較的容量が大きく，描画するにも時間がかかる場合があります．有償のものでは，小容量，高速表示を特長とするメーカ独自のフォントや，描画ライブラリも提供されています．

さらに，組み込み機器を使用する地域や想定されるユーザによっては，各国のフォントを搭載して切り替え使用が必要な場合もあります．組み込み機器の場合，文字表示には注意すべき点がいくつかあります．

Column AG903 対応の組み込み Linux「Axell Linux」とは？

「Axell Linux」は，アクセル社が提供する Yocto ベースの組み込み Linux です．現在，同社製 SoC「AG903」に対応しています．Yocto は組み込み機器に合わせたカスタマイズが容易で，多くの組み込み Linux ベンダに採用されています．GUI に Qt の利用も可能で，多種多様な製品に採用される AG903 には最適な Linux 開発環境と言えるでしょう．同社が提供する Non-OS や RTOS 向けの SDK とソースコードの共通化も可能で，Linux への移行もスムーズに行えるのも特徴です．

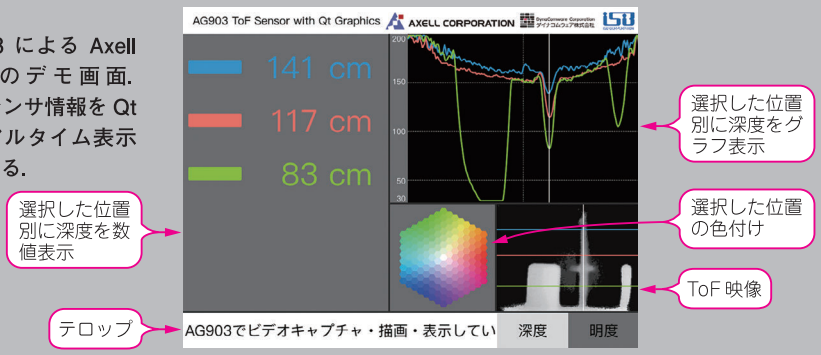


図 A
AG903 による Axell Linux のデモ画面．ToF センサ情報を Qt でリアルタイム表示している．

第3章 AG903を用いたGUIの実現方法

本章では, AG903 を用いた GUI の実現方法の事例を紹介します.

組み込み GUI では，ターゲット機器によって実現すべき機能やハードウェアのリソースが大きく異なります. いくつかのツールベンダから GUI の開発環境と実行環境が提供されていますが, ベンダ依存の部分が多くなるとともに, 用途によって一長一短があります. OS ベースの GUI のように標準的, 汎用的な環境になってはいません.

ここではすべての事例について, プログラミングで実現するとして, どのような制御を行えばよいかを解説します. 複数の制御方法があれば, それらを比較して解説していきます.

3-1 画面の構成

図 3-1 のように，表示したい画像を VRAM 内に用意しておけば，それぞれ表示位置とサイズを指定して, 1 つの画面に合成して表示することができます. VRAM 内の画像は, AG903 内蔵の各回路ブロックを用いて生成できます. 画像ごとに部分的に透過したり，アルファブレンドしたりすることも可能です.

表示画面にボタンや背景を配置して表示させる方法には 2 通りあります.

(1) 表示回路 (DSP) で画面構成する方法

AG903 の表示回路はウィンドウ表示機能を備えていますので，このウィンドウ表示機能を利用して画面を構成できます (**図 3-2**).

VRAM 格納位置, サイズ, 表示位置, 表示優先順位等を指定して各画像にウィンドウを割り当て，表示回路でウィンドウ合成して表示します. **表 3-1** に，ウィンドウ表示機能の設定項目を示します. 合成時に 1 画素単位で透過させたり，アルファブレンディングすることも可能です. 透過は, 透過色を指定するか, 画素のアルファ値を 0 にすることで実現できます.

ウィンドウによる画面構成は，VRAM 使用量，使用帯域，CPU 負荷のいずれも最小にできる画面構成方法です. CPU は，表示に必要な最低限の情報のみを設定するだけで画面構成できます. ただし，合成できるのは矩形領域のみとなります. 表示可能なウィンドウ数は 16 で，それを超える場合は描画回路と併用する必要があります. また，拡大 / 縮小表示も可能ですが，その場合は VRAM の使用帯域が増えるため, 等倍で表示させるのが基本です.

これらの制約はありますが, VRAM の使用量、使用帯域を削減させるため, 工夫してなるべくこの方法に持ち込むのが良いでしょう. AG903 の表示機能に関しては、トラ技小冊子「ビデオ接続規格と画像表示の勘所」も参考にしてください.

図 3-1　画像合成の例

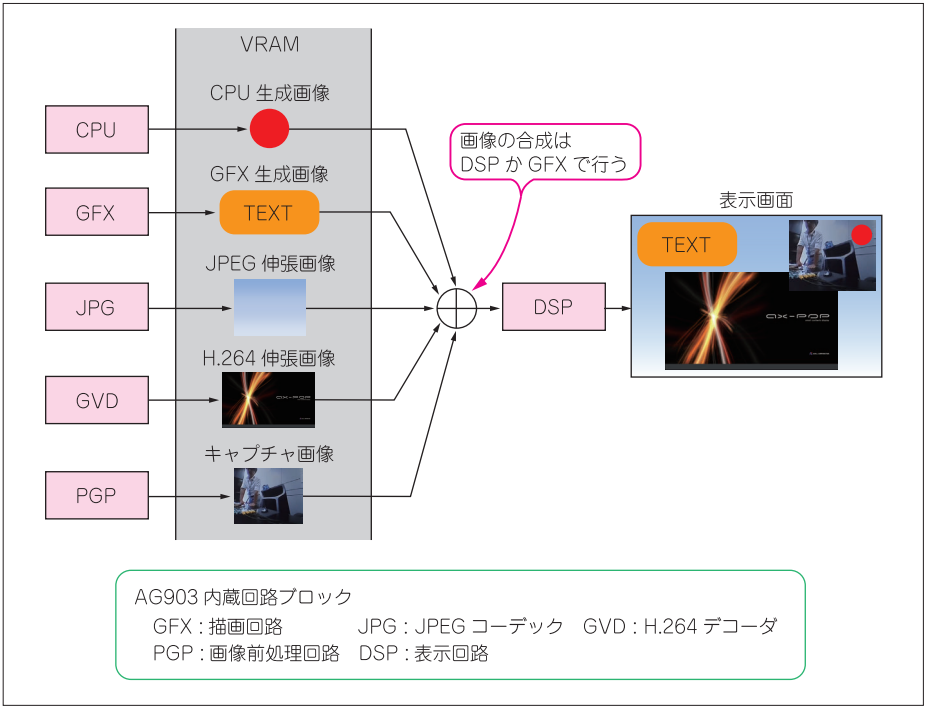


図 3-2　表示回路（DSP）で画像合成する例

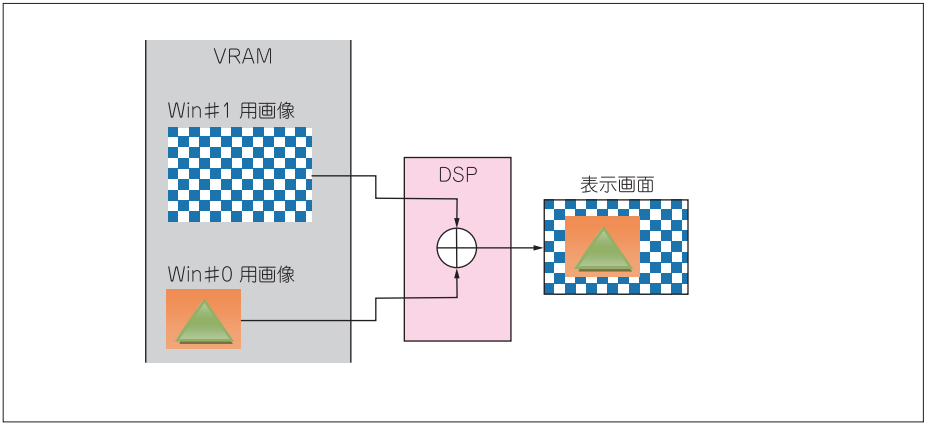


表 3-1　表示回路（DSP）のウィンドウ設定内容

表示位置, サイズ	VRAM 画像位置, サイズ	VRAM ストライド幅
拡大縮小方法	水平 / 垂直反転表示 ON/OFF	表示 ON/OFF
カラーフォーマット	VRAM パレット位置	パレットカラー
標準 α 値	透過色（青, 緑, 赤, α）	透過色判定方法
データスワップ方法		

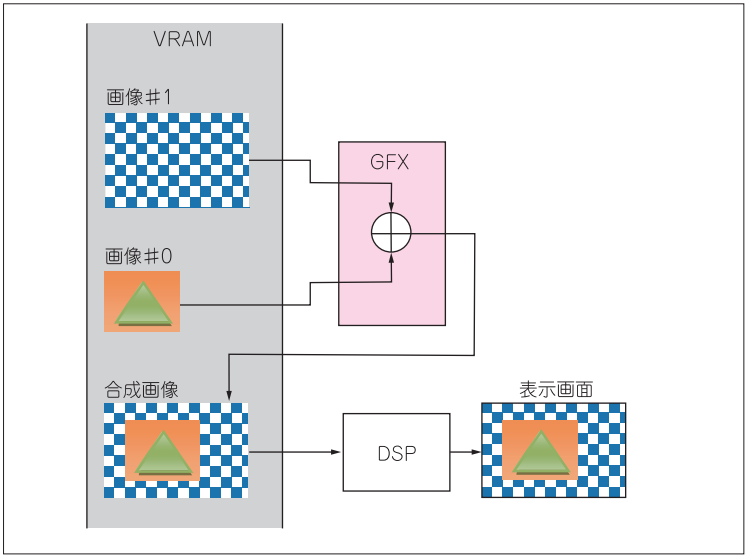
(2) 描画回路（GFX）で画面構成する方法

描画回路で VRAM 内に合成画像を生成し, それを表示回路で表示します（図 3-3）.

この方法は, 合成時に画像の拡大, 縮小, 回転ができることや, フォグなどの多彩なエフェクトをかけることができます. 表示回路と同じで, 合成時に 1 画素単位で透過させたり, アルファブレンディングすることもできます. 合成する画像の数や領域も任意に指定できます. 最終的に, 合成した画像をウィンドウに割り当てることで表示できます.

なお, 描画回路による画像合成については, 4-4 章 p.32 と 4-9 章 p.61 にも解説がありますので, そちらも参考にしてください.

図 3-3　描画回路（GFX）で画像合成する例



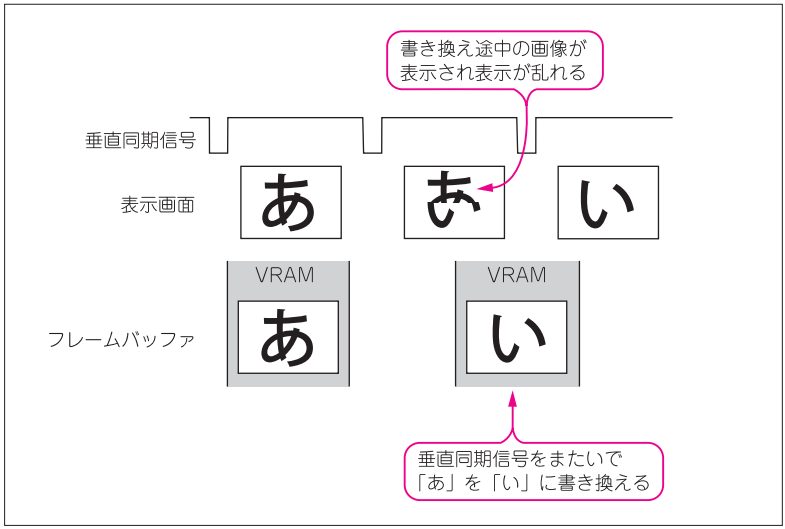
3-2 画面の更新

表示回路は，1 画面を構成する画像データを VRAM（フレームバッファ）から読み出し，ドットクロック，水平同期信号，垂直同期信号に同期してビデオ信号フォーマットに変換して出力します．垂直同期信号の 1 周期ごとに，1 画面分のデータが読み出されます．

このとき，VRAM の読み出し先を変更しない限り，毎回同じ VRAM 領域を読み出して同じ内容を表示します．読み出し先を変更すれば，表示内容もそれに応じて変わります．

ビデオの同期信号を無視して表示内容の書き換えを行うと，**図 3-4** に示すように，書き換え途中の画像が一旦表示されてしまい，見る側に違和感を与える場合があります．

図 3-4　表示期間中の画像書き換えによる表示乱れの例



表示の乱れを生じさせずに画像を更新する方法として，2 つの方法を紹介します．

(1) ダブルバッファ

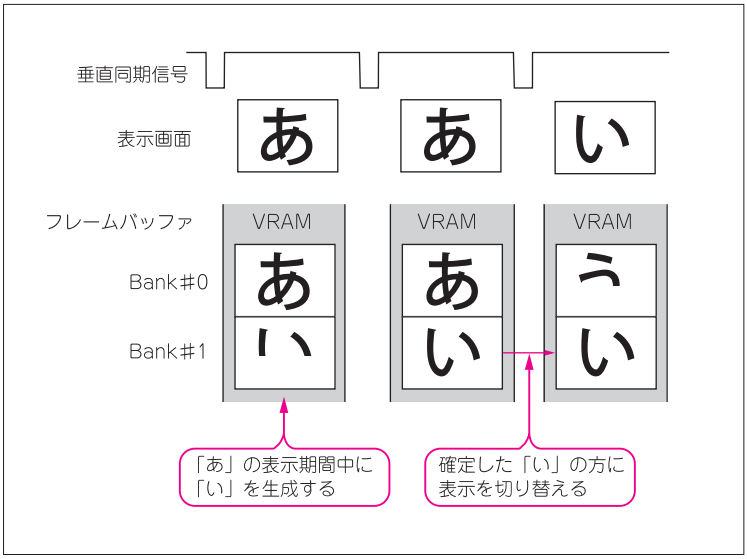
VRAM に 2 つのフレームバッファ（ダブルバッファ）を用意し，表示用と更新用交互に切り替えて使用します（**図 3-5**）．

AG903 はウィンドウ表示機能を備えていますので，このバッファは画面全体ではなく，ウィンドウごとに設けることも可能です．バッファの更新完了を待ってから VRAM の読み出し先を切り替えれば，表示が乱れることはありません．表示回路の VRAM 読み出し先の更新は垂直同期信号のタイミングで行われるので，実際の表示の更新は，次の垂直同期信号のタイミングで行われます．

ダブルバッファ方式には，表示中にバッファの更新が行えるので，更新処理に対する時間的制約が緩和されるという大きな利点があります．一方，バッファを 2 個にするので VRAM 使用量が増えるのは難点です．

シングルバッファで表示の乱れが生じない方法として，ブランキング中に画像を更新するという方法もあります．しかし，更新処理に使える時間が短いことや，さらにドットクロック周波数を低下させる目的でブランキング期間を設けない場合もありますので，ダブルバッファによる方式が一般的となってます．

図 3-5　ダブルバッファによる表示更新の例



(2) 画像更新中の表示をオフする

AG903 は，画面全体の表示をオン / オフすることもできますが，ウィンドウごとに表示をオンオフすることもできます．そこで，画面上，部分的に表示をオフして，この期間に画像データを更新します．実際の表示オン / オフは垂直同期信号に同期して行われるので，表示オフしてから次の垂直同期信号を待って，それ以降に画像データを更新する必要があります．

部分的にオフした画像の下に背景画像を重ねて配置しておけば，表示をオフした期間にも背景画像は変わらず表示されるので，オフした部分は目立たず，画面のデザイン性を保つことができます．

この方法は，画像データ 1 面分のバッファのみを確保すればよいため，VRAM の使用量を削減できます．

3-3 画面の遷移

画面の遷移時に，表示をぱっと切り替える代わりに，徐々に変化させていくなどの効果をもたせることができます．この画面の遷移は，3-1 章「画面の構成」と 3-2 章「画面の更新」の応用編と言えます．

画面遷移中の表示画面の作成は，3-1 章で説明したように，表示回路による方法または描画回路による方法で実現できます．また，画面遷移の操作によって表示画面に乱れを生じないように，3-2 章で説明したように，垂直同期信号を意識した操作が必要になります．

ここでは，表示回路を用いた 2 つの画面遷移方法について説明します．

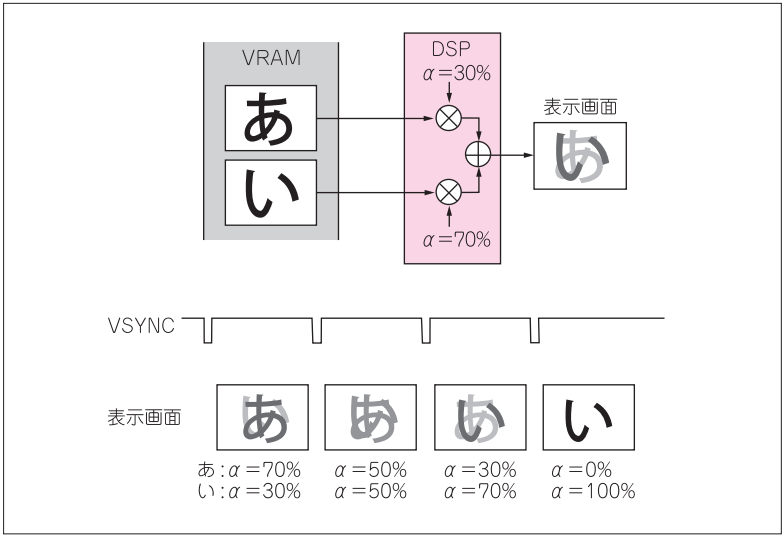
(1) クロスフェード

画面遷移時に，現在表示をフェードアウトさせ，それに重ねて次の表示をフェードインさせる効果です．

現在表示用の画像と，次に表示する画像の 2 画像を用意し，画面上の同じ位置に表示させます．ただし，垂直同期信号（VSYNC）のタイミングごとにそれぞれのアルファ値を設定し，アルファブレンドして画面遷移させます（図 3-6）．

表示回路によるアルファブレンドは 256 段階の設定が可能です．

図 3-6　クロスフェードの例



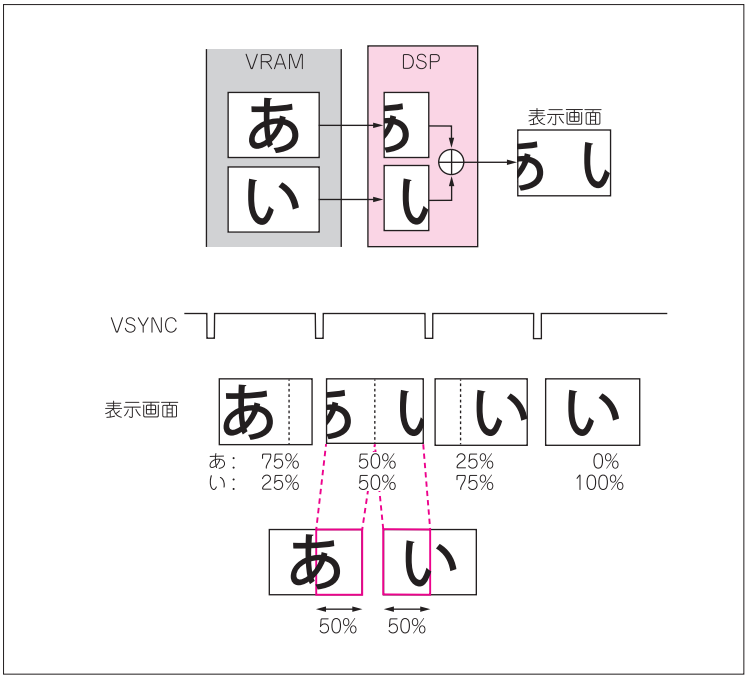
(2) ワイプ

画面遷移時に，現在表示をシフトさせて画面外に追い出し，それに続けて次の表示をシフトさせて画面内に導き入れる効果です．

現在表示用の画像と，次に表示する画像の 2 画像を用意し，垂直同期信号（VSYNC）のタイミングごとにそれぞれ VRAM の読み出し位置と表示する領域，サイズを変えて画面遷移させます（図 3-7）．

ワイプもクロスフェードと同様に，DSP への設定で実現できる画面遷移方法で，CPU の処理負荷は高くありません．ただし，クロスフェードは遷移期間中，VRAM からの読み出しが 2 画面分必要になるのに対して，ワイプの方は合わせて 1 画面分のみで良いため，VRAM の使用帯域としてはクロスフェードよりも低くできます．画面遷移については，CPU の処理負荷に合わせて VRAM の使用帯域も考慮する必要があります．

図 3-7　ワイプの例



3-4 ボタン操作とアニメーション表示

アニメーション表示を行う場合，各フレームの画像をあらかじめ VRAM に作成しておくと，その後の処理が容易になります．

図 3-8 に，ボタンアニメーションの実現例を示します．これは，アニメーションのフレーム数が 11 フレームある例です．

#0 のフレームは「カメラ」ボタンが選択されていない状態を表し，#10 のフレームは「カメラ」ボタンが選択されている状態を表します．画面上の「カメラ」ボタン付近をタッチすることで，垂直同期信号に同期して，#0 から #10 まで表示する画像を順に切り替えます．

ボタンのように小さい画像の合成は，描画回路で行います．あらかじめアニメーションフレームを作成しておけば，表示する画像を選択する操作を行うだけで，アニメーション表示できます．

各フレームの画像は，透過色，またはアルファ値を用いることで，多様な背景に合成できます．**図 3-8** の例では，丸ボタンの外側をアルファ値で透過した画像を作成しています．

図 3-8　ボタンアニメーションの実現例

3-5 テロップ表示

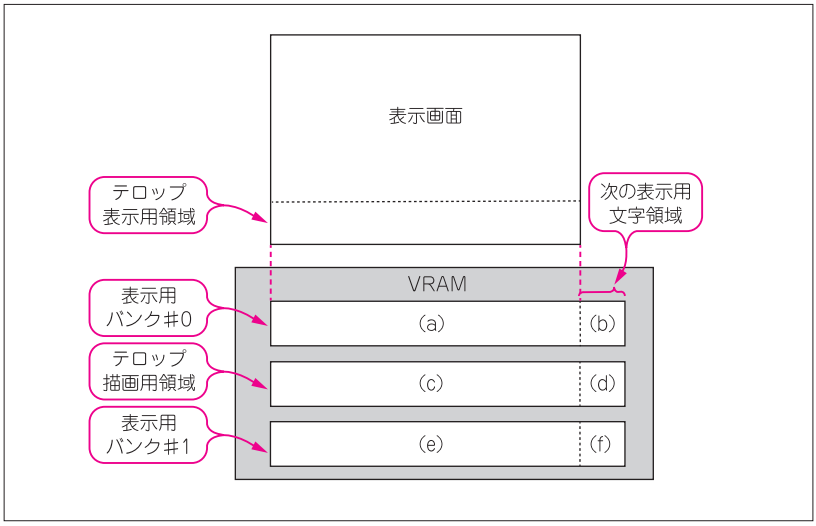
テロップ表示は，画面に表示しきれない文字列を，一定の方向に移動しながら表示させる方法です．ニュース情報など任意のテキストを表示することを想定すると，アニメーション表示のように事前に画像を用意することは適していません．また，確実に垂直同期信号に同期して文字を移動させないと，いわゆる「カクツキ」という現象に感じとられてしまいます．

垂直同期信号の周期は概ね 16.7msec になりますが，文字の描画は比較的時間がかかりますので，時間的な制約を考慮した工夫が必要になります．

ここでは，ウィンドウ表示機能を利用した方法を紹介します．文字を移動させずに，ウィンドウを移動させる方法です．

図 3-9 のように，画面に表示するテロップと同じサイズの領域に 1 文字分足して，3 つの領域を VRAM に確保します．(c)＋(d) は文字列の描画用の領域で，描画結果を (a)＋(b) または (e)＋(f) に交互にコピーして，(a)＋(b) と (e)＋(f) をダブルバッファとして表示を繰り返します．描画と表示の座標系が異なる前提で，ここでは 3 つの領域を確保しています．

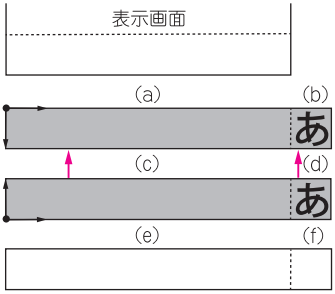
図 3-9　テロップ表示のためのフレームバッファ構成例



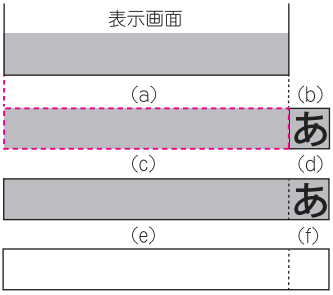
この確保した領域を使って 2 文字のテロップ表示を行う手順例を，**図 3-10** の①～⑥に示します．

図 3-10　テロップ表示の手順例

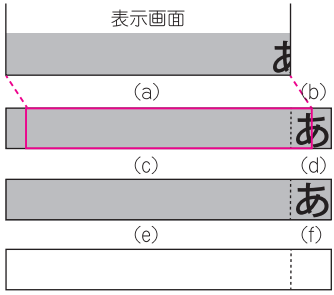
①(d)の領域に 1 文字描画して，(c)+(d)を(a)+(b)の領域に Y 軸反転コピーします．描画の座標系はボトムアップとし，表示の座標系はトップダウンとしています．



②次の垂直同期信号のタイミングで (a) の領域をウィンドウで表示させます．



③垂直同期信号のタイミングで (a)+(b) 内で表示する領域を移動させます．ウィンドウの移動になりますので，移動の最小単位は 1 画素になります．(b)までウィンドウを移動させれば，1 文字分の表示が終わります．



④ウィンドウを移動させている間に，描画用領域に次の文字を描画します．(c)+(d) の内容を 1 文字分左にずらしてコピーし，空いた (d) に新しい文字を描画します．その後，(c)+(d) を (e)+(f) にコピーします．

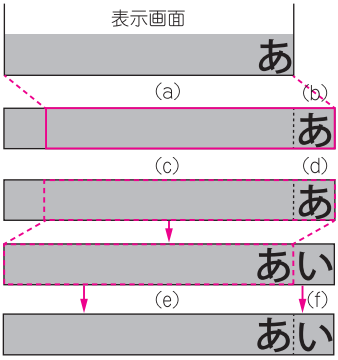
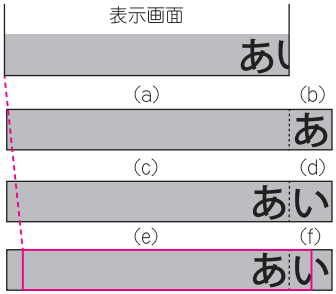
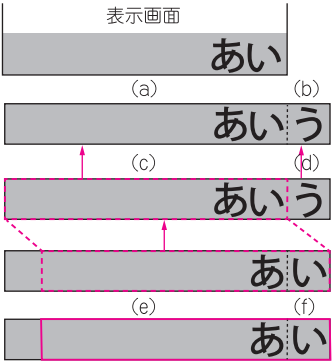


図 3-10　テロップ表示の手順例（続き）

⑤手順③と同様に，垂直同期信号のタイミングで (e)＋(f) 内で表示する領域を移動させます．最初の表示位置は，1 移動単位分ずらした位置になります．(f) までウィンドウを移動させれば，2 文字目までの表示が終わります．



⑥ウィンドウを移動させている間に，描画用領域に次の文字を描画します．④と同じく，(c) ＋(d) の内容を 1 文字分左にずらしてコピーし，空いた (d) に新しい文字を描画します．その後，(c)＋(d) を (a)＋(b) にコピーします．



この後は③〜⑥を繰り返すことによって，続けて文字列を表示できます．
また，同様の方法で，縦方向に移動するテロップも実現することができます．

第4章 AG903内蔵描画機能の操作方法

本章では，AG903が内蔵している描画回路を用いてグラフィックスを描画する方法について解説します．この描画回路は OpenVG に準拠したハードウェアアクセラレータです．まずOpenVGの概要を簡単に紹介し，その後に描画の手順，さまざまな描画の方法をコード例とともに説明していきます．

4-1 OpenVGとは

OpenVGは，2次元ベクタグラフィックスおよびラスタグラフィックス用のAPIです．非営利の技術コンソーシアムである Khronos グループが策定し，2005年に OpenVG 1.0，2008年にOpenVG 1.1の仕様が公開されました．AG903は，OpevVG 1.1および EGL 1.3規格に準拠しています．

OpenVGのAPIは，線やベジェ曲線などの単純なジオメトリに基づいた2次元描画で定義しており，SVG(Scalable Vector Graphics)フォーマットの画像やフォントの描画などを行う際に，ハードウェアによるアクセラレーションが可能です．アンチエイリアス処理もサポートされており，高品質に画像の拡大，縮小，回転などの変形が行えます．

モバイル機器，ゲーム機，カーナビゲーション，家電製品，デジタルカメラ，医療機器などのユーザインターフェースで多く利用されています．仕様策定から10年以上経過しても仕様修正，改版などは行われておらず，長期利用が前提の産業分野でも安心して利用できるグラフィックス用のAPIです．

OpenVG はプラットフォーム非依存であることが特長です．プラットフォーム依存の拡張仕様を使用していなければ，PC上で描画結果を確認したソースを，AG903用に再コンパイルすることで，AG903でも同様の結果を得られます．

OpenVGの規格書は，Khronosグループのホームページから入手可能です．

https://www.khronos.org/openvg/

→ [Specification] →「1.1-Specification」→「openvg_1_1_spec.pdf」

また，OpenVGのサンプルソース(Windows版)も同ホームページから入手可能です．

→ [Resouces] →「OpenVG 1.1 Sample Implementation (December 1, 2008)」

Khronos グループとは

グラフィックス，並列計算などのAPIに関する無償でオープンな規格の策定・普及を目的とする団体です．3次元グラフィックス用 API の OpenGL，並列計算用 API の OpenCL，2次元ベクタグラフィックス用の OpenVG などの規格で知られています．

4-2 AG903 内蔵 OpenVG アクセラレータの使い方

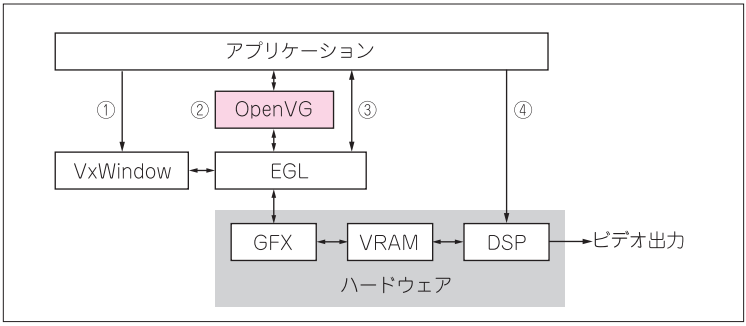
AG903 内蔵の描画回路(GFX)は，OpenVG 準拠のアクセラレータです. ここでは OpenVG を利用したグラフィックス描画の手順について説明します.

プラットフォーム非依存である OpenVG を AG903 のハードウェア上で実行するために，ハードウェア依存部を何らかの方法で吸収することが必要です. Khronos グループでは，さまざまなハードウェア上で OpenVG などの描画 API を利用可能にするために，ハードウェアごとの差異を吸収するインターフェース(API)として，EGL を定義しています. EGL は，OpenGL，OpenGL ES などでも使用できます.

アクセル社が AG903 用として提供しているグラフィックス描画ライブラリは，この EGL を使用しています.

OpenVG による描画の前後で，EGL の操作が必要になります. **図 4-1** に示す①〜④の順に，基本的な操作手順を説明します.

図 4-1　AG903 内蔵 OpenVG アクセラレータによるグラフィックス描画



① 初期化

EGL には，描画結果の表示用に，プラットフォーム固有のディスプレイとウィンドウが必要になります. AG903 のライブラリでは，これらを仮想的に生成する API が用意されています.

次のコードは，32 ビットカラーで横 640× 縦 480 画素のウィンドウを 1 つ生成します.

複数のウィンドウが必要な場合は，VXCreateWindow をウィンドウ数だけ呼び出して VXWindow オブジェクトを複数生成する必要があります.

```
VXDisplay *vx_display = VXOpenDisplay(NULL);
VXWindow vx_window = VXCreateWindow(vx_display, 0, 0, 640, 480, 32);
```

必要な数のウィンドウが生成できたら，EGL を使用する前に，次のコードのようにして EGL を初期化します．

```
EGLDisplay egl_display;

egl_display = eglGetDisplay(vx_display);
eglInitialize(egl_display, NULL, NULL);
eglBindAPI(EGL_OPENVG_API);
```

次に，eglChooseConfig により，各色要素のビット数などの条件を指定して，描画先の条件に合う EGL のコンフィグレーション（EGLConfig）をシステムのプリセット値から選択します．次のコードは，XRGB8888 用のコンフィグレーションを選択する例です．

```
EGLint config_attr[] = {
  EGL_CONFIG_ID, 9,    /* XRGB8888 用コンテキストを選択 */
  EGL_NONE
};
EGLConfig egl_config;
int num_config;

eglChooseConfig(egl_display, config_attr, &egl_config, 1, &num_config);
```

eglChooseConfig は，指定された条件に最も合うコンフィグレーションを所定のルールに従ってプリセット値から検索しますが，ARGB8888 や XRGB8888 などカラーフォーマットのビット数が一致するものもあって，必ずしも期待するコンフィグレーションが得られない場合があります．そこで，eglChooseConfig で複数のコンフィグレーションを取得してから所望のコンフィグレーションを選択するという方法も取ることはできますが，検索条件に EGL_CONFIG_ID を指定することで，確実に期待するコンフィグレーションを選択することもできます．

表 4-1 に EGL_CONFIG_ID とコンフィグレーションの関係の一例を示します．なお，EGL_CONFIG_ID に対する EGL のコンフィグレーション内容は，プラットフォームに依存します．この表は AG903 用です．

ここで選択するコンフィグレーションが持つカラーフォーマットのビット数は，生成した仮想ウィンドウのカラーフォーマットのビット数と一致している必要があります．そのため，使用したいカラーフォーマットに合わせたビット数で仮想ウィンドウを生成しておきます．

OpenVG 内部で処理される画素は sRGB 色空間＋アルファ値で管理されますが，最終的に EGL のコンフィグレーションで選択したカラーフォーマットに変換してフレームバッファに格納されます．

表 4-1　AG903 で定義される EGL_CONFIG_ID とコンフィグレーションの例

EGL_CONFIG_ID	カラーフォーマットのビット数	アルファマスクバッファ
1	16 ビット(RGB565)	なし
2	16 ビット(RGB565)	4 ビット
3	16 ビット(ARGB1555)	なし
4	16 ビット(ARGB1555)	4 ビット
5	16 ビット(ARGB4444)	なし
6	16 ビット(ARGB4444)	4 ビット
7	32 ビット(ARGB8888)	なし
8	32 ビット(ARGB8888)	8 ビット
9	32 ビット(XRGB8888)	なし
10	32 ビット(XRGB8888)	8 ビット
11	8 ビット(A8)	なし
12	8 ビット(A8)	8 ビット
13	8 ビット(L8)	なし
14	8 ビット(L8)	8 ビット
15	1 ビット(BW1)	なし
16	1 ビット(BW1)	1 ビット
31	16 ビット(YCbCr422)	なし
32	16 ビット(YCbCr422)	4 ビット
33	24 ビット(RGB888)	なし
34	24 ビット(RGB888)	8 ビット

表の「A8」は，アルファ値だけが格納されます.「L8」は，8 ビットの輝度値として格納されます(CLUT8).「BW1」は,1 ビットの輝度値として格納されます(CLUT1).

次に，eglCreateContext により，eglChooseConfig で選択したコンフィグレーションに対応した描画コンテキスト(EGLContext)を生成します.

複数のウィンドウが必要な場合は，ウィンドウ数だけ描画コンテキストを生成する必要があります.

```
EGLContext egl_context;

egl_context = eglCreateContext(
                egl_display, &egl_config, EGL_NO_CONTEXT, NULL);
```

次に，描画先の条件（下記コードの surface_attr）を指定して，eglCreateWindowSurface で，描画先のバッファ（フレームバッファ）を生成します．OpenVG では，このバッファを描画サーフェス（EGLSurface）と呼びます．

EGL_RENDER_BUFFER は，バッファ方式を指定します．EGL_BACK_BUFFER でダブルバッファに，EGL_SINGLE_BUFFER でシングルバッファになります．

EGL_VG_ALPHA_FORMAT は，描画サーフェスのアルファ値の扱いを指定します．OpenVG の描画パイプラインは，アルファ値が RGB 値に乗算された状態（Pre-Multiply）で処理されます．描画サーフェスに画素値を格納する際，Pre-Multiply のまま格納する場合は，EGL_VG_ALPHA_FORMAT_PRE を指定します．一方，RGB 値からアルファ値を割って Non-Premultiply にして格納する場合は，EGL_VG_ALPHA_FORMAT_NONPRE を指定します．

EGL_VG_COLORSPACE は，描画サーフェスの色空間を指定します．ガンマ補正された色空間（Non-Linear）を指定する場合は，EGL_VG_COLORSPACE_sRGB を指定します．リニアな色空間を指定する場合は，EGL_VG_COLORSPACE_LINEAR を指定します．

複数のウィンドウが必要な場合は，ウィンドウ数だけ描画サーフェスの生成が必要です．

```
EGLint surface_attr[] = {
  EGL_RENDER_BUFFER, EGL_BACK_BUFFER,
  EGL_ALPHA_FORMAT,  EGL_ALPHA_FORMAT_NONPRE,
  EGL_COLORSPACE,    EGL_COLORSPACE_sRGB,
  EGL_NONE
};
EGLSurface egl_surface;
egl_surface = eglCreateWindowSurface(
                egl_display, &egl_config, vx_window, surface_attr);
```

最後に，eglMakeCurrent を実行して，生成した描画サーフェスと描画コンテキストを使用可能な状態にします．これにより，OpenVG の処理が実行可能になります．

複数のウィンドウを使用する場合は，OpenVG による描画前に，ウィンドウごとの描画サーフェスと描画コンテキストを eglMakeCurrent で都度切り替える必要があります．

なお，OpenVG の各オブジェクトは，描画コンテキストに関連付けられていますので，描画コンテキストを切り替えると切り替え前のオブジェクトにはアクセスできなくなります．基本的には，描画コンテキスト切り替え前後の描画処理は独立したものとするのが良いでしょう．

```
eglMakeCurrent(egl_display, egl_surface, egl_surface, egl_context);
```

②描画

OpenVG の API を使用して描画します. 描画の完了は vgFinish で待ちます.

```
VGfloat ClearColor[4]  = { 0.0f, 0.0f, 0.0f, 1.0f };
vgSetfv(VG_CLEAR_COLOR, 4, ClearColor);
vgClear(0, 0, 640, 480);
⋮
vgFinish();
```

③バッファフリップ

eglSwapBuffers により, 表示する描画サーフェスの更新を行います.

複数のウィンドウを使用する場合, ウィンドウごとの描画サーフェスを eglSwapBuffers で更新する必要があります.

```
eglSwapBuffers(egl_display, egl_surface);
```

④表示

生成した仮想ウィンドウは, そのままでは表示されません. フレームバッファのアドレスや表示位置などを表示回路（DSP）に設定する必要があります. ウィンドウサイズやカラーフォーマットは生成した仮想ウィンドウと同じ条件を先に DSP に設定しておきます.

eglSwapBuffers 実行後, eglQuerySurface に EGL_FRONT_BUFFER_ADDRESS_EXT を指定して, 更新したバッファの先頭アドレスを取得し, DSP に設定することで実際の表示が更新されます.

複数のウィンドウを使用する場合, 同様の手順をウィンドウごとに行う必要があります.

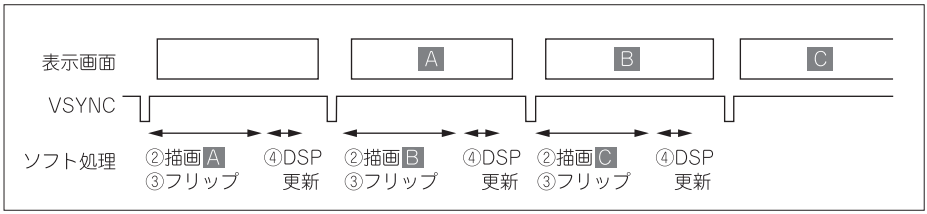
```
EGLint addr;
eglQuerySurface(
  egl_display, egl_surface, EGL_FRONT_BUFFER_ADDRESS_EXT, &addr);

dsp_set_fbase(GM2M(addr));  /* DSP の FBASE に設定 */
dsp_wait_vsync();                /* VSYNC 待ち */
```

※ dsp_set_fbase と dsp_wait_vsync は OpenVG の API ではありません.

次画面の描画の実行は, DSP に設定した内容が確実に反映された後に行います. そのため, 垂直同期信号のタイミングを待った後, 手順②から行います（**図 4-2**）. ④で更新されたフレームバッファは, 次の垂直同期信号の後に画面に表示されていきます.

以降, 手順②〜④を繰り返して画面を更新していきます.

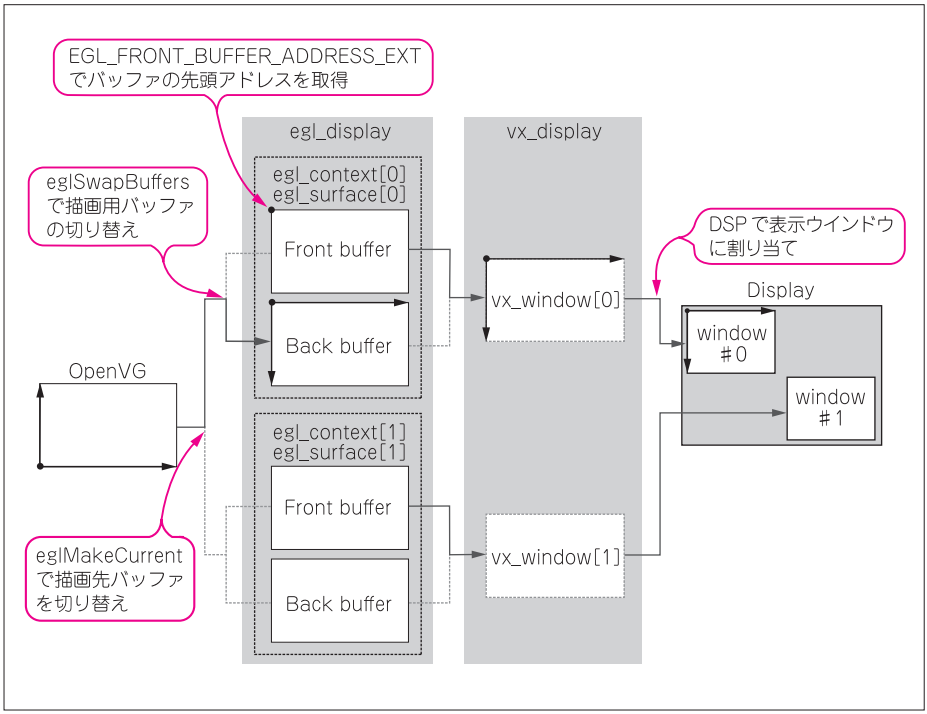
図 4-2　フレームバッファの更新と表示タイミング



以上の手順について，2 つのウィンドウを使用する場合の各オブジェクトの関係を図 4-3 に示します．

なお，OpenVG の座標系は左下が原点になります（ボトムアップ）．これに対し，DSP の座標系は左上が原点になります（トップダウン）．実際に描画が行われる際は Y 座標を反転する操作が行われるため，EGL_FRONT_BUFFER_ADDRESS_EXT を指定して得られるバッファの先頭アドレスをそのまま DSP に設定しても，OpenVG による描画イメージのまま表示が行われます．OpenVG の座標系は左下が原点ということを念頭に描画を行う必要があります．

図 4-3　2 つのウィンドウ使用時におけるオブジェクトの関係



4-3 OpenVG の座標系

OpenVG では，パス，画像，文字，塗りつぶし用ペイント，ストローク用ペイントおよび描画サーフェス，それぞれに座標系があります．

パス，画像，文字の座標系を単にユーザ座標系と呼びます．

パス，画像，文字，塗りつぶし用ペイント，ストローク用ペイントの各座標系は，最終的に，行列によって描画サーフェスの座標系に変換します．vgSeti で VG_MATRIX_MODE を指定して設定できるマトリクスモードは，**表 4-2** に示す 5 種類があります．行列を操作する API を使用する前に，適切なマトリクスモードを設定しておくことが必要です．

表 4-2　VG_MATRIX_MODE を指定して設定できるマトリクスモード

VG_MATRIX_PATH_USER_TO_SURFACE	パスを描画サーフェス座標に変換
VG_MATRIX_IMAGE_USER_TO_SURFACE	画像を描画サーフェス座標に変換
VG_MATRIX_GLYPH_USER_TO_SURFACE	文字を描画サーフェス座標に変換
VG_MATRIX_FILL_PAINT_TO_USER	塗りつぶし用ペイントをユーザ座標に変換
VG_MATRIX_STROKE_PAINT_TO_USER	ストローク用ペイントをユーザ座標に変換

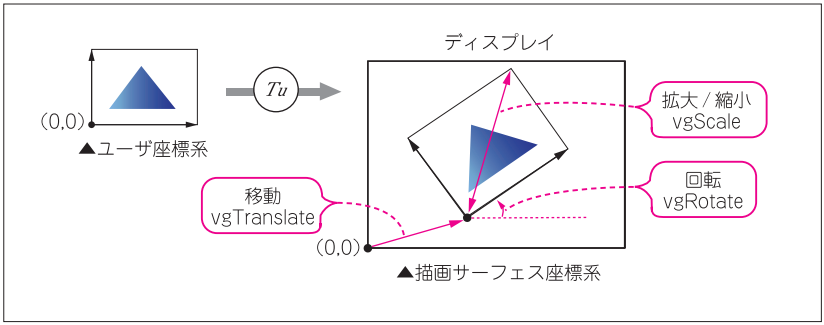
行列による座標変換は，アフィン変換です．

拡大縮小（vgScale），回転（vgRotate），せん断（vgShear），および平行移動（vgTranslate）の API が用意されていて，任意の組み合わせが可能です．

これらの API は，現在の行列に対して右乗算されます．行列の積は非可換ですので，期待する結果になるよう実行順序を意識する必要があります．行列による操作がどのように行われるかは，API の実行順とは逆に見ていくと理解しやすいです．

OpenVG によるパス，画像，文字は，それぞれのユーザ座標系にマッピングされます．ユーザ座標系から描画サーフェス座標系へは，アフィン変換して描画されます（**図 4-4**）．

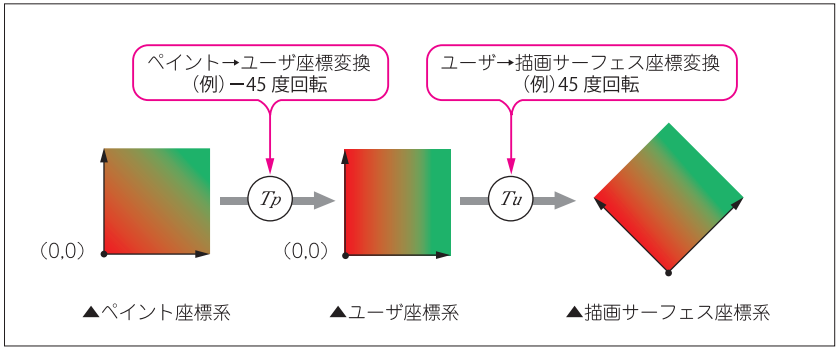
図 4-4　ユーザ座標系から描画サーフェス座標系への変換例



各ユーザ座標系のマトリクスが vgLoadIdentity により単位行列に設定されていれば，そのユーザ座標系は描画サーフェス座標系と同一と見なせます．

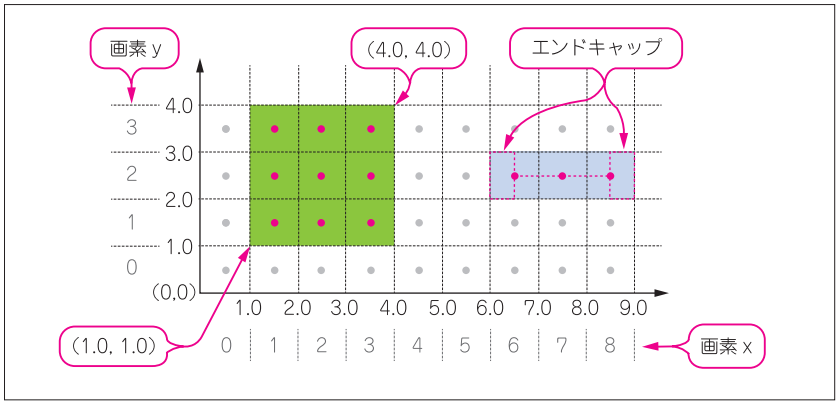
ペイントは独自の座標系（ペイント座標系）を持ちます．一度ペイント座標系からユーザ座標系にマッピングされ，その後描画サーフェス座標系に変換されます（図 4-5）．

図 4-5　ペイント座標系から描画サーフェス座標系への変換例



OpenVG の座標は画素単位ではありません．OpenVG は，画素（x, y）の中心が（x+0.5, y+0.5）にあります．この中心が描画する図形に含まれるかどうかで，実際の描画が処理されます．図 4-6 の（1.0, 1.0）から（4.0, 4.0）の矩形領域を緑色で塗りつぶす操作は，画素（1, 1）から（3, 3）の塗りつぶしになります．線幅 1 画素のラインは，指定座標から線の方向垂直に ±0.5 広げて塗りつぶされます．図 4-6 の始点（6.5, 2.5）から終点（8.5, 2.5）の青色ラインは，正方形のエンドキャップを付けて画素（6, 2）から（8, 2）のラインになります．

図 4-6　座標の指定と塗りつぶし範囲の例



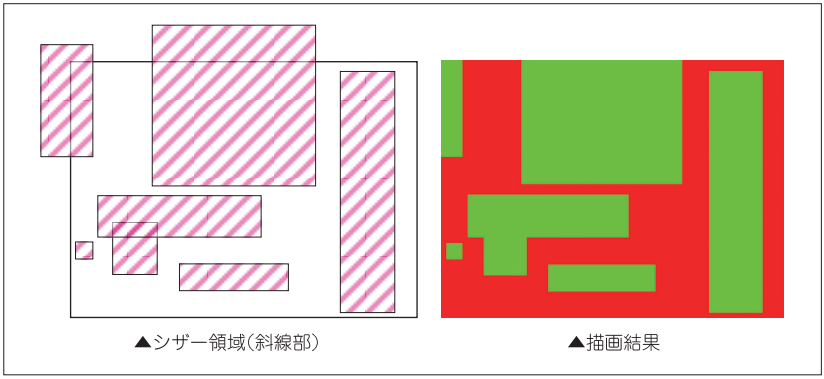
4-4 OpenVG による描画領域の指定

領域を指定して描画する方法として，シザー処理とアルファマスクがあります．

シザー領域を指定すると，以降のすべての描画は，指定の矩形範囲内にある画素のみが描画されるようになります．シザー領域は，最大 32 か所指定できます．

図 4-7 にシザー領域を指定した場合の描画例を示します．最初に画面全体を赤色で描画し，次にシザー領域を指定して画面全体を緑色で描画します．緑色の全画面描画は，シザーで指定した矩形領域内のみ描画されます．

図 4-7　シザー領域を 7 か所指定した場合の描画例



シザー領域の指定は，VG_SCISSOR_RECTS を指定して vgSetiv で設定します．1 つのシザー領域は，始点の左下座標（x, y），幅，高さの 4 要素で構成されますので，4 の倍数の要素数を持った 1 次元配列を指定します．指定したシザー領域を有効 / 無効にするには，VG_SCISSORING を指定して vgSeti で設定します．シザー処理を有効にする場合は VG_TRUE を，無効にする場合は VG_FALSE を設定します．

```
static const VGint ScissorRects[] = {
   80, 80,  80, 80,      } シザー領域 2 か所の指定例
  200, 50, 200, 50       } 1 か所あたり x, y, 幅 , 高さの順で指定
};

vgSetiv(VG_SCISSOR_RECTS,
        sizeof(ScissorRects)/sizeof(ScissorRects[0]), ScissorRects);
vgSeti(VG_SCISSORING, VG_TRUE);
```

アルファマスク機能を使うと，1 画素単位で任意形状での型抜きや，任意比率での背景合成ができます．**図 4-8** のように，アルファマスクバッファにマスクパターンを生成して，文字に合わせた型抜きやアルファ値を施した背景合成が行えます．

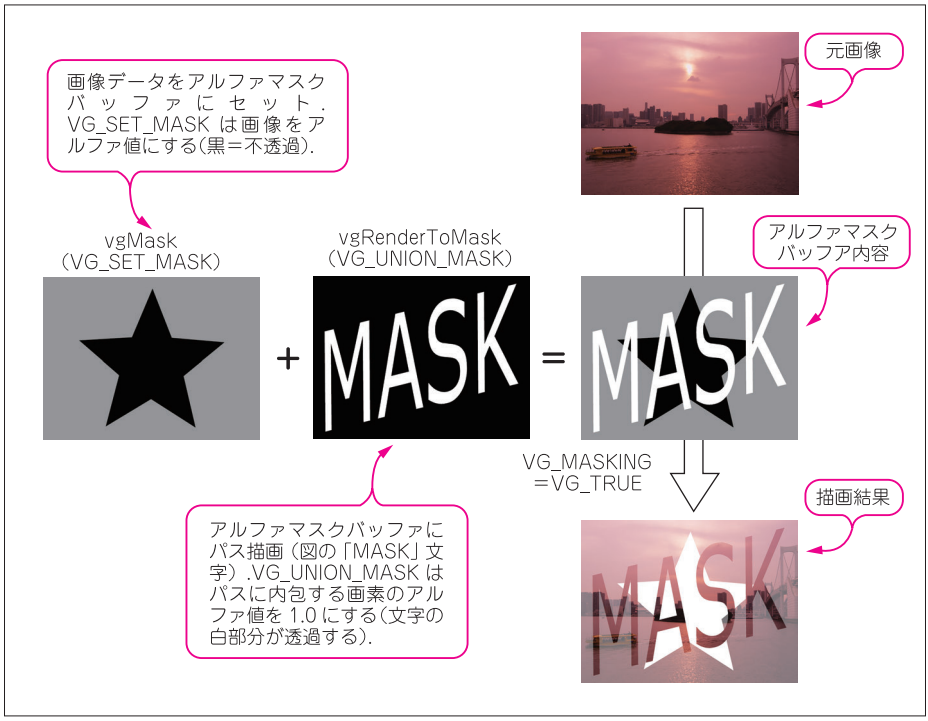
アルファマスクバッファに VGImage オブジェクトを格納する場合は vgMask を使います．アルファ値を持つ画像の場合はアルファ値が，アルファ値を持たない画像の場合は R 値がアルファマスクバッファに格納されます．

アルファマスクバッファに VGPath オブジェクトを描画する場合は vgRenderToMask を使います．

アルファマスク処理を有効 / 無効にするには，VG_MASKING を指定して vgSeti で設定します．アルファマスク処理を有効にする場合は VG_TRUE を，無効にする場合は VG_FALSE を設定します．

なお，アルファマスクが使えるかどうかは，EGL のコンフィグレーションによって異なります．そのため，初期化時の eglChooseConfig において，アルファマスクバッファをサポートするコンフィグレーションを選択しておく必要があります．

図 4-8　アルファマスクの例

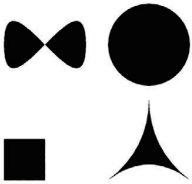


4-5 OpenVG による描画例

OpenVG の API を利用した基本的な描画例を示します. AG903 では, 独自の拡張命令により, YCbCr422 色空間を使用したり, キャプチャ画像をダイレクトに扱うこともできます.

描画サーフェスのクリア

任意の矩形領域を任意の色で塗りつぶすことができます.

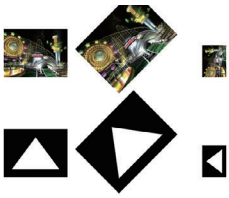


パス描画

直線、2 次ベジェ曲線、3 次ベジェ曲線、楕円弧の 4 種類を組み合わせてパスを形成できます. また, パスで指定した領域内を塗りつぶすこともできます.

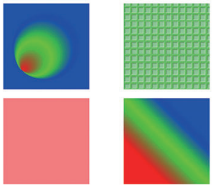
ストローク描画

ライン幅、エンドキャップ形状、ジョイン形状、ダッシュパターン等を指定して, パスに沿ってストローク描画することができます.



イメージ描画とマトリクス

2 次元のラスタデータを扱うことができます. また, アフィン変換により, 拡大／縮小, 回転等を行って描画することができます.



ペイント

円形グラデーション（左上：グラデーションの半径と焦点を指定), パターンペイント(右上：画像を指定), カラーペイント(左下：色を指定), 線形グラデーション(右下：カラーテーブルを指定)による画像の修飾ができます.

以下で, これらの描画のそれぞれの例を示して解説します.

(1) 描画サーフェスのクリア

TestClearSurface は，描画サーフェスを横方向に 4 分割し，左から赤，緑，青，灰色でクリアします．

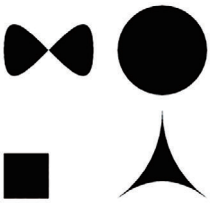
クリアカラーは VG_CLEAR_COLOR を指定して vgSetfv で設定します．

カラー値は，R 値，G 値，B 値，アルファ値それぞれ 0.0 ～ 1.0 の範囲で設定します．R 値，G 値，B 値の 1.0 は最大輝度を意味し，アルファ値の 1.0 は不透過を意味します．同じ色で続けてクリアする場合は，再設定は不要です．

実際のクリア処理は，位置と範囲を指定して vgClear で行います．

```
static void ClearSurface(VGfloat r, VGfloat g, VGfloat b, VGfloat a,
  VGint x, VGint y, VGint width, VGint height)
{
  VGfloat color[4];
  color[0] = r;
  color[1] = g;
  color[2] = b;
  color[3] = a;

  /* クリアカラーを設定 */
  vgSetfv(VG_CLEAR_COLOR, 4, color);

  /* 指定された領域をクリア */
  vgClear(x, y, width, height);
}

void TestClearSurface(int width, int height)
{
  VGint clearwidth = width / 4;

  /* 画面を横方向に4分割し、左から赤、緑、青、灰色でクリアする */
  ClearSurface(1.0f, 0.0f, 0.0f, 1.0f, clearwidth * 0, 0, clearwidth, height);
  ClearSurface(0.0f, 1.0f, 0.0f, 1.0f, clearwidth * 1, 0, clearwidth, height);
  ClearSurface(0.0f, 0.0f, 1.0f, 1.0f, clearwidth * 2, 0, clearwidth, height);
  ClearSurface(0.5f, 0.5f, 0.5f, 1.0f, clearwidth * 3, 0, clearwidth, height);
}
```

(2) パスによる描画



パスによる描画は，始点の移動、直線、2 次または 3 次のベジェ曲線、楕円弧を描く各コマンドを複数組み合わせて実現します．基本的には，MOVE_TO コマンドから始めて MOVE_TO コマンドまたは CLOSE_PATH コマンドで終了します．

各コマンドには，いくつかのパラメータを持つものがあります．

パスの描画時に，パスに沿って描画するか，パスの内側を塗りつぶすか，またはその両方を行うか指定します．

表 4-3 に OpenVG のパス描画のコマンド一覧を示します．

表 4-3　OpenVG パス描画のコマンド一覧

コマンド動作	コマンド名	パラメータ	コマンド実行後の座標
前の終点から始点まで直線で結ぶ	CLOSE_PATH	なし	(px,py)=(ox,oy) =(sx,sy)
始点を (x0,y0) に移動する	MOVE_TO	x0,y0	(sx,sy)=(px,py) =(ox,oy)=(x0,y0)
前の終点から (x0,y0) まで直線で結ぶ	LINE_TO	x0,y0	(px,py)=(ox,oy) =(x0,y0)
前の終点から x0 まで水平線で結ぶ	HLINE_TO	x0 [y0=oy]	(px,py)=(x0,oy) ox=x0
前の終点から y0 まで垂直線で結ぶ	VLINE_TO	y0 [x0=ox]	(px,py)=(ox,y0) oy=y0
(x0,y0) を制御点として，前の終点から (x1,y1) まで 2 次ベジェ曲線で結ぶ	QUAD_TO	x0,y0,x1,y1	(px,py)=(x0,y0) (ox,oy)=(x1,y1)
(x0,y0)，(x1,y1) を制御点として，前の終点から (x2,y2) まで 3 次ベジェ曲線で結ぶ	CUBIC_TO	x0,y0, x1,y1, x2,y2	(px,py)=(x1,y1) (ox,oy)=(x2,y2)
前の終点から (x1,y1) まで前の曲線と滑らかに接続する 2 次ベジェ曲線で結ぶ	SQUAD_TO	x1,y1 [(x0,y0) =(2×ox-px, 2×oy-py)]	(px,py) =(2×ox-px, 2×oy-py) (ox,oy)=(x1,y1)

表 4-3　OpenVG パス描画のコマンド一覧（続き）

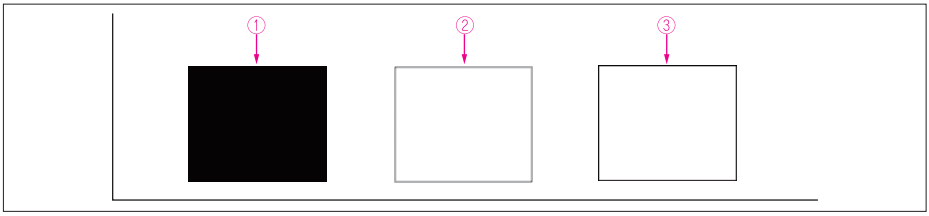
コマンド動作	コマンド名	パラメータ	コマンド実行後の座標
(x1,y1) を制御点として，前の終点から (x2,y2) まで前の曲線と滑らかに接続する 3 次ベジェ曲線で結ぶ	SCUBIC_TO	x1,y1,x2,y2 [(x0,y0) = (2×ox-px, 2×oy-py)]	(px,py)=(x1,y1) (ox,oy)=(x2,y2)
水平半径 rh，垂直半径 rv，回転角 rot の楕円弧で前の終点から (x0,y0) まで反時計方向の小回りで結ぶ	SCCWARC_TO	rh,rv,rot, x0,y0	(px,py)=(ox,oy) =(x0,y0)
水平半径 rh，垂直半径 rv，回転角 rot の楕円弧で前の終点から (x0,y0) まで時計方向の小回りで結ぶ	SCWARC_TO	rh,rv,rot, x0,y0	(px,py)=(ox,oy) =(x0,y0)
水平半径 rh，垂直半径 rv，回転角 rot の楕円弧で前の終点から (x0,y0) まで反時計方向の大回りで結ぶ	LCCWARC_TO	rh,rv,rot, x0,y0	(px,py)=(ox,oy) =(x0,y0)
水平半径 rh，垂直半径 rv，回転角 rot の楕円弧で前の終点から (x0,y0) まで反時計方向の大回りで結ぶ	LCWARC_TO	rh,rv,rot, x0,y0	(px,py)=(ox,oy) =(x0,y0)

(sx,sy)　：現在のパスの始点．つまり最後の MOVE_TO コマンド指定の位置．初期値は (0,0)．
(ox,oy)　：前のコマンドによる終点．初期値は (0,0)．
(px,py)　：コマンドが 2 次または 3 次ベジェ曲線の場合は，前のコマンドによる最後の制御点，
それ以外は前のコマンドによる終点．初期値は (0,0)．

次に，パス描画を用いて図形を描画する例を見てみましょう．

下記にコードを示す TestDrawRect は，**図 4-9** のように，サイズ 200×100 の長方形を 3 種類描画します．

図 4-9　TestDrawRect の実行結果（画面左下部分のみを示します）




```
void TestDrawRect(void)
{
  VGubyte cmd[] = {
    VG_MOVE_TO_ABS, VG_LINE_TO_ABS, VG_LINE_TO_ABS, VG_LINE_TO_ABS,
    VG_CLOSE_PATH
   };
   VGshort coord[] = { 50, 50,  50, 150,  170, 150,  170, 50 };
   VGPath path;

  /* 利用可能な最高速度で描画する */
  vgSeti(VG_RENDERING_QUALITY, VG_RENDERING_QUALITY_FASTER);

  /* VGPath オブジェクトの生成（16bit 整数） */
  path = vgCreatePath(VG_PATH_FORMAT_STANDARD, VG_PATH_DATATYPE_S_16,
                      1.0f, 0.0f, 0, 0, VG_PATH_CAPABILITY_ALL);

  /* VGPath オブジェクトに形状データを登録 */
  vgAppendPathData(path, sizeof(cmd), cmd, coord);

  /* パスの変換マトリクスを設定 */
  vgSeti(VG_MATRIX_MODE, VG_MATRIX_PATH_USER_TO_SURFACE);

  /* マトリクスを初期化（単位行列） */
  vgLoadIdentity();

  /* ① VGPath を描画（塗りつぶし） */
  vgDrawPath(path, VG_FILL_PATH);
```

```
    /* 移動行列指定 */
    vgTranslate(170.0f, 0.0f);

    /* ストロークの幅を設定 */
    vgSeti(VG_STROKE_LINE_WIDTH, 1);

    /* ② VGPath を描画(ストローク) */
    vgDrawPath(path, VG_STROKE_PATH);

    /* 移動行列指定 */
    vgTranslate(170.5f, 0.5f);

    /* ③ VGPath を描画(ストローク) */
    vgDrawPath(path, VG_STROKE_PATH);

    /* VGPath オブジェクトを破棄 */
    vgDestroyPath(path);
}
```

パスの描画を行うためには, vgCreatePath で生成した VGPath オブジェクトが必要です. 第 2 引数で, 座標データの型(精度)を 8 ビット整数(VG_PATH_DATATYPE_S_8), 16 ビット整数 (VG_PATH_DATATYPE_S_16), 32 ビット整数 (VG_PATH_DATATYPE_S_32), IEEE754 浮動小数点(VG_PATH_DATATYPE_F)から選択して指定します.

8 ビット整数は, 指定できる座標の範囲が -128 ～ +127 であることに注意が必要です. フォントなどのように比較的狭い範囲でパスが構成される図形では, データ容量を節約できるメリットがあります.

第3引数 scale と第4引数 bias により, 入力座標値 v は、(scale × v + bias)として解釈されます。通常は, scale=1.0f, bias=0.0f として, 入力座標値そのもので描画するようにしておきます.

使い終わった VGPath オブジェクトは, vgDestroyPath で必ず破棄します.

vgDestroyPath は, 描画サーフェスの更新(eglSwapBuffers)を待つことなく, vgDrawPath 実行後, その VGPath オブジェクトを使用することがなければ, すぐに実行することができます.

```
path = vgCreatePath(VG_PATH_FORMAT_STANDARD,VG_PATH_DATATYPE_S_16,
                    1.0f, 0.0f, 0, 0, VG_PATH_CAPABILITY_ALL);
vgAppendPathData(path, sizeof(cmd), cmd, coord);
```

VGPath オブジェクトに vgAppendPathData でパスを追加します. 複数のコマンドを配列にして一括で追加することができます.

第2引数にコマンド数, 第3引数にコマンドの配列へのポインタ, 第4引数にコマンドに対するパラメータの配列へのポインタをそれぞれ指定します. コマンドとそのパラメータは, 配列を分けてそれぞれ1次元の配列で指定することになります.

なお, 座標の指定は, パスの座標系(ユーザ座標系)内で絶対値または相対値で指定できます. **表 4-3** に示したコマンドの末尾に「_ABS」を付けると座標の指定は絶対座標になり, 「_REL」を付けると前のコマンドの終点からの相対座標指定になります. どちらも, 描画サーフェス座標系での座標指定ではなく, ユーザ座標系での指定になります.

パラメータの数は, コマンドによって異なります.

パラメータの型は, vgCreatePath で指定した精度に合わせる必要があります. 16 ビット整数(VG_PATH_DATATYPE_S_16)とした場合は VGshort とします.

TestDrawRect で実行する下記のコマンドは, **図 4-10** に示すパスを作成します.

```
VGubyte cmd[] = {
  VG_MOVE_TO_ABS,          ← 絶対座標指定で始点の移動
  VG_LINE_TO_ABS,          ← 絶対座標指定の直線(a)でパスを結ぶ
  VG_LINE_TO_ABS,          ← 絶対座標指定の直線(b)でパスを結ぶ
  VG_LINE_TO_ABS,          ← 絶対座標指定の直線(c)でパスを結ぶ
  VG_CLOSE_PATH            ← 始点まで直線でパスを結ぶ
};
VGshort coord[] = {
    50,  50,               ← 始点の絶対座標を x0, y0 順で指定
    50, 150,               ← 直線(a)の終点の絶対座標を x0, y0 順で指定
   170, 150,               ← 直線(b)の終点の絶対座標を x0, y0 順で指定
   170,  50,               ← 直線(c)の終点の絶対座標を x0, y0 順で指定
};
```

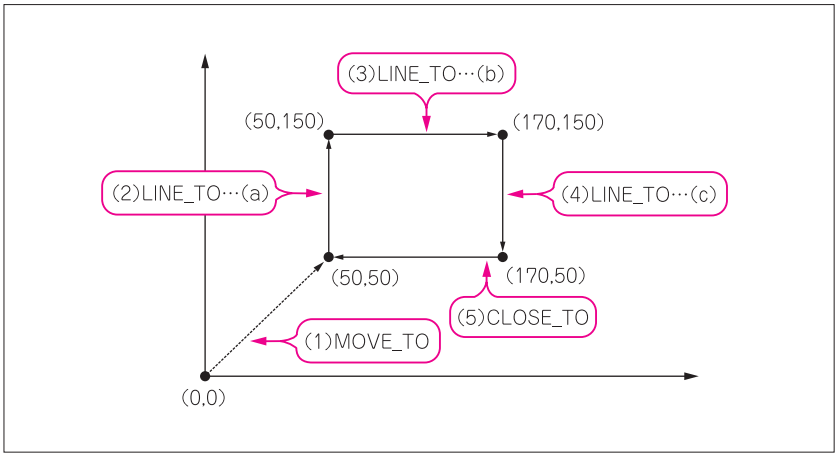
パスの描画前に, パスの座標系(ユーザ座標系)から描画サーフェス座標系に変換する行列を初期化しておきます.

vgSeti で VG_MATRIX_MODE を指定して, VG_MATRIX_PATH_USER_TO_SURFACE を設定することで, 以降の行列操作はパスを描画サーフェス座標系に変換する行列に対して行われます.

ここでは, vgLoadIdentity により単位行列に設定します.

```
vgSeti(VG_MATRIX_MODE, VG_MATRIX_PATH_USER_TO_SURFACE);
vgLoadIdentity();
```

図 4-10　TestDrawRect で実行するパスの接続過程



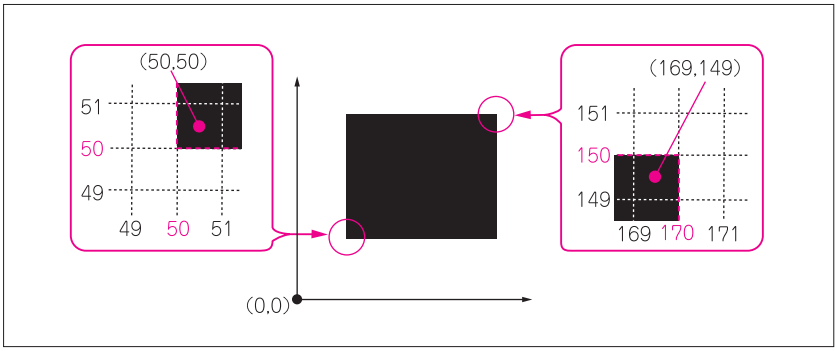
生成したパスは, vgDrawPath で描画します(**図 4-11**).

第 2 引数にはパスの描画方法を指定します. VG_FILL_PATH を指定すると, パスで指定された領域に含まれる画素が塗りつぶされます.

TestDrawRect では, パスの左下頂点の座標が(50, 50), 右上頂点は(170, 150)を指定していますので, 画素単位で見ると左下位置(50, 50), 右上位置(169, 149)の横 120× 縦 100 画素の塗りつぶされた長方形が描画されます. パスそのものに幅はありません.

```
vgDrawPath(path, VG_FILL_PATH);
```

図 4-11　TestDrawRect の塗りつぶし長方形①の描画



次に，長方形の枠だけを描画（ストローク描画）してみます．vgDrawPath の第 2 引数に VG_STROKE_PATH を指定すると，パスに沿った線のみが描画されます．

線幅は，整数で設定する場合は vgSeti で，浮動小数点で設定する場合は vgSetf で VG_STROKE_LINE_WIDTH を指定して設定します．線幅 1 の指定は，線幅 1 画素の指定に相当します．

描画先は，前の塗りつぶした長方形から右方向に 170 画素だけずれた位置に描画します．座標データで直接ずらした位置の座標を指定することもできますが，同一形状であればパスデータを再利用できます．

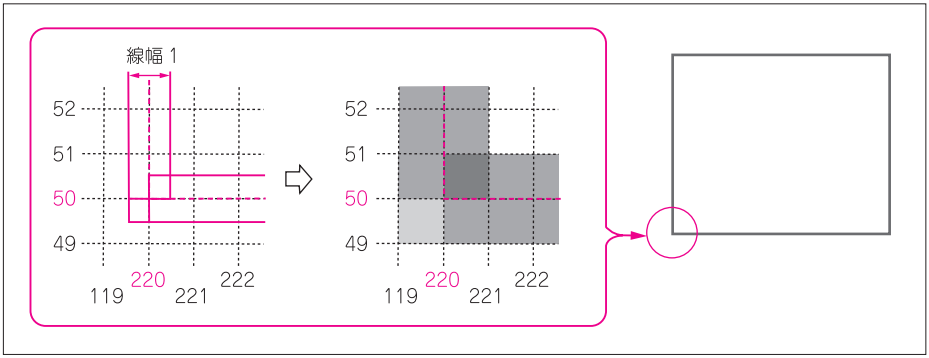
vgTranslate で描画先の原点を (170, 0) だけ相対移動（座標変換）します．長方形の元の座標が左下頂点 (50, 50)，右上頂点 (170, 150) のパスですので，これは，同じく絶対座標で見ると左下頂点 (50+170, 50+0)→(220, 50)，右上頂点 (170+170, 150+0)→(340, 150) のパスを指定したのと等価です．

このように同じ画像を部品として使用することを想定し，パスデータの座標系（ユーザ座標系）と実際の描画先の座標系（描画サーフェス座標系）は分かれています．これにより，パスデータの作成時は，実際に描画される位置やサイズなどを意識する必要がなくなります．ユーザ座標系は，行列変換を用いて実際の描画サーフェス座標系にマッピングされます．

ところで，今回の長方形は画素単位で見ると線幅 2 の灰色の線に見えます（**図 4-12**）．これは，パスが座標軸上を指定されており，線幅は 1 画素であるため，実際に描画する線が隣り合う画素を含んでいるためです．結果として，アンチエイリアス処理により，黒色が各画素に配分されて灰色となります．

```
vgTranslate(170.0f, 0.0f);
vgSeti(VG_STROKE_LINE_WIDTH, 1);
vgDrawPath(path, VG_STROKE_PATH);
```

図 4-12　TestDrawRect の長方形②のストローク描画



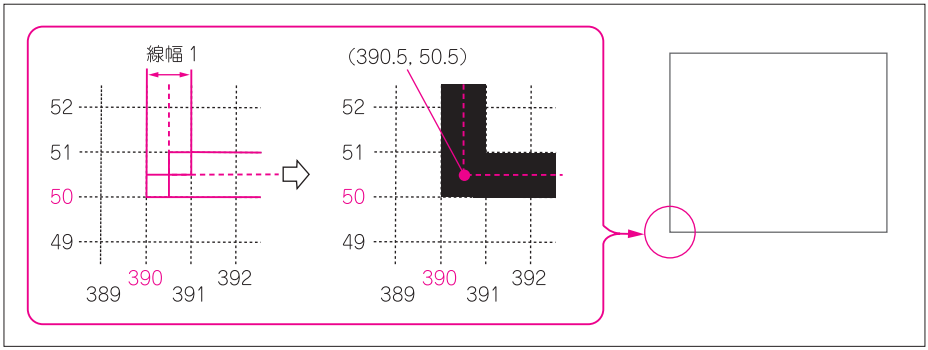
次に，画素の中心を通過するようなパスを指定して長方形のストローク描画を行います．

vgTranslate で描画先の原点を更に（170.5, 0.5）だけ相対移動します．長方形の元の座標に対して絶対座標で見ると，並行移動を 2 回行っていますので，左下頂点（50+170+170.5, 50+0+0.5）→（390.5, 50.5），右上頂点（170+170+170.5, 150+0+0.5）→（510.5, 150.5）のパスを指定したのと等価です．結果として，線幅 1 画素の線で長方形が描画されます（**図 4-13**）．

画素単位で所望の図形を描画するには，1 画素の大きさを意識する必要があります．

```
vgTranslate(170.5f, 0.5f);
vgDrawPath(path, VG_STROKE_PATH);
```

図 4-13　TestDrawRect の長方形③のストローク描画

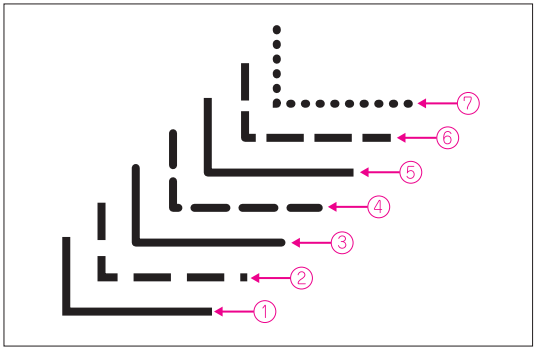


(3) ストローク描画

次に，ストローク描画の例を見てみましょう．ストローク描画は，線幅、エンドキャップ形状、ジョイン形状、ダッシュパターンが指定できます．

次にコードを示す TestDrawLine は，**図 4-14** のように線を 7 種類描画します．

図 4-14　TestDrawLine の実行結果



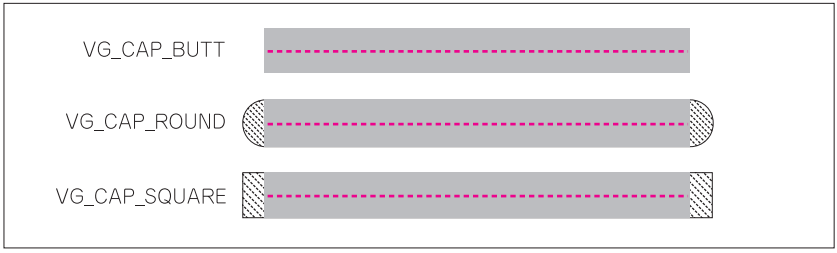
```
static void DrawLine(VGPath path, VGint lineWidth, VGCapStyle cap, VGJoinStyle join,
  VGint dashCount, VGint *dashPattern, VGfloat x, VGfloat y)
{
  /* ストロークの幅を設定 */
  vgSeti(VG_STROKE_LINE_WIDTH, lineWidth);

  /* エンドキャップ形状の設定 */
  vgSeti(VG_STROKE_CAP_STYLE, cap);

  /* ジョイン形状の設定 */
  vgSeti(VG_STROKE_JOIN_STYLE, join);

  /* ダッシュパターンの設定 */
  vgSetiv(VG_STROKE_DASH_PATTERN, dashCount, dashPattern);

  /* 指定した位置にストローク描画 */
  vgLoadIdentity();
  vgTranslate(x, y);
  vgDrawPath(path, VG_STROKE_PATH);
}

void TestDrawLine(void)
{
  VGubyte cmd[] = { VG_MOVE_TO_ABS, VG_LINE_TO_ABS, VG_LINE_TO_ABS };
  VGint coord[] = { 0, 100, 0, 0, 200, 0 };
  VGint DashPattern0[] = { 40, 30 };
  VGint DashPattern1[] = {  0, 20 };
  VGPath path;

  /* 利用可能な最高速度で描画する */
  vgSeti(VG_RENDERING_QUALITY, VG_RENDERING_QUALITY_FASTER);

  /* 現在のマトリクスをパス変換マトリクスに設定 */
  vgSeti(VG_MATRIX_MODE, VG_MATRIX_PATH_USER_TO_SURFACE);

  /* VGPath オブジェクトの生成(32bit 整数) */
  path = vgCreatePath(VG_PATH_FORMAT_STANDARD, VG_PATH_DATATYPE_S_32,
                      1.0f, 0.0f, 0, 0, VG_PATH_CAPABILITY_ALL);

  /* VGPath オブジェクトに形状データを登録 */
  vgAppendPathData(path, sizeof(cmd), cmd, coord);
```

```
    /* ① 線幅 15pixel, キャップ =BUTT, ジョイン =MITER, ダッシュ処理なし */
    DrawLine(path, 15, VG_CAP_BUTT, VG_JOIN_MITER, 0, NULL,          50,  50);
    /* ② 線幅 15pixel, キャップ =BUTT, ジョイン =MITER, ダッシュ処理あり */
    DrawLine(path, 15, VG_CAP_BUTT, VG_JOIN_MITER, 2, DashPattern0,      100, 100);
    /* ③ 線幅 15pixel, キャップ =ROUND, ジョイン =ROUND, ダッシュ処理なし */
    DrawLine(path, 15, VG_CAP_ROUND,  VG_JOIN_ROUND, 0, NULL,          150, 150);
    /* ④ 線幅 15pixel, キャップ =ROUND, ジョイン =ROUND, ダッシュ処理あり */
    DrawLine(path, 15, VG_CAP_ROUND,  VG_JOIN_ROUND, 2, DashPattern0,     200, 200);
    /* ⑤ 線幅 15pixel, キャップ =SQUARE, ジョイン =BEVEL, ダッシュ処理なし */
    DrawLine(path, 15, VG_CAP_SQUARE, VG_JOIN_BEVEL, 0, NULL,          250, 250);
    /* ⑥ 線幅 15pixel, キャップ =SQUARE, ジョイン =BEVEL, ダッシュ処理あり */
    DrawLine(path, 15, VG_CAP_SQUARE, VG_JOIN_BEVEL, 2, DashPattern   0, 300, 300);
    /* ⑦ 線幅 15pixel, キャップ =ROUND, ジョイン =ROUND, ダッシュ処理あり */
    DrawLine(path, 15, VG_CAP_ROUND,  VG_JOIN_ROUND, 2, DashPattern1,     350, 350);

    /* VGPath オブジェクトを破棄 */
    vgDestroyPath(path);
}
```

線幅は，整数で設定する場合は vgSeti で，浮動小数点で設定する場合は vgSetf で VG_STROKE_LINE_WIDTH を指定して設定します．線幅 1 の指定は，線幅 1 画素の指定に相当します．

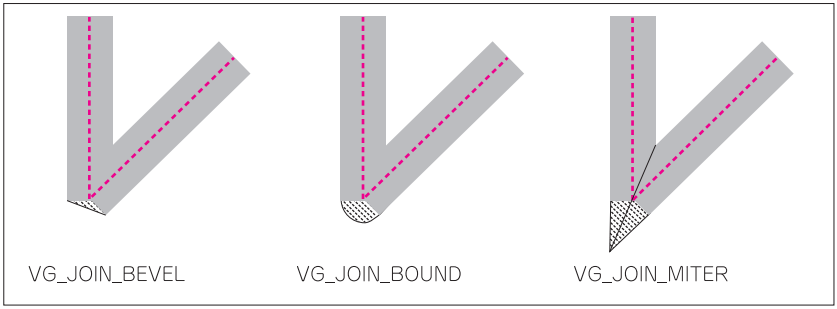
エンドキャップ形状は vgSeti で，VG_STROKE_CAP_STYLE を指定して設定します．VG_CAP_BUTT はキャップなし，VG_CAP_ROUND は半円，VG_CAP_SQUARE は四角形を描画します（**図 4-15**）．

図 4-15　エンドキャップ形状（破線はパス）



ジョイン形状は vgSeti で，VG_STROKE_JOIN_STYLE を指定して設定します．VG_JOIN_BEVEL は線の結合部を直線にして埋めます．VG_CAP_ROUND は線の結合部を円形に丸めて埋めます．VG_JOIN_MITER は線の結合部を鋭角にして埋めます（**図 4-16**）．

図 4-16　ジョイン形状（破線はパス）



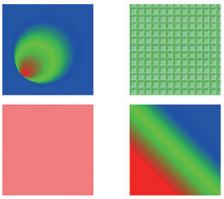
ダッシュパターン（破線，点線など）は vgSetiv で，VG_STROKE_DASH_PATTERN を指定して設定します．ダッシュパターンは ON 区間（画素を生成する部分の長さ）と OFF 区間（画素を生成しない部分の長さ）のペアで指定します．最大 16 セグメント（8 つの ON/OFF ペア）のダッシュパターンが指定可能です．このダッシュパターンは，vgSetfv を使用することで浮動小数点値でも設定可能です．

このストローク描画例で，ダッシュパターン DashPattern0 を使用するストローク描画（②，④，⑥）は，ON 区間の長さが 40，OFF 区間の長さ 30 のダッシュパターンを繰り返して描画します．

ダッシュパターンを無効化する場合（①，③，⑤）は，VG_STROKE_DASH_PATTERN にサイズ 0 の配列を指定します（vgSetiv あるいは vgSetfv の第 2 引数に 0 を指定する）．

⑦のストローク描画例もダッシュパターンを有効にして描画していますが，特殊な使い方の例になります．ダッシュパターン DashPattern1 で ON 区間の長さが 0，OFF 区間の長さを 20 に指定しています．この場合，ON 区間が無いためラインの描画は行われません．しかし，エンドキャップ形状に VG_CAP_ROUND を指定しているため，エンドキャップの描画だけが行われ，結果として丸い点線が描画されます．

(4) ペイント

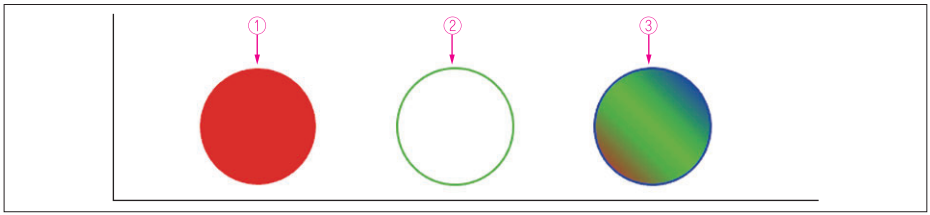


パスの描画には，カラーペイント（色を指定），線形グラデーション（カラーテーブルを指定），円形グラデーション（グラデーションの半径と焦点を指定），パターンペイント（画像を指定）による修飾が行えます．

この修飾は，塗りつぶし描画とストローク描画に分けて指定できます．

次に示すコードは，DrawCircle で円形のパスを作成し，TestDrawPaintCircle で 3 種類のペイントを行った例です（**図 4-17**）．

図 4-17　TestDrawPaintCircle の実行結果（画面左下部分のみを示します）




```
static void DrawCircle(VGfloat x, VGfloat y, VGbitfield paintModes)
{
  VGubyte cmd[] = { VG_MOVE_TO_ABS, VG_LCCWARC_TO_ABS, VG_LCCWARC_TO_ABS };
  VGfloat coord[] = {
  100.0f, 50.0f,
  50.0f, 50.0f, 0.0f,  100.0f, 150.0f,
  50.0f, 50.0f, 0.0f,  100.0f, 50.0f
  };
  VGPath path;

  /* VGPathオブジェクトの生成(浮動小数点) */
  path = vgCreatePath(VG_PATH_FORMAT_STANDARD, VG_PATH_DATATYPE_F,
                      1.0f, 0.0f, 0, 0, VG_PATH_CAPABILITY_ALL);

  /* VGPathオブジェクトに形状データを登録 */
  vgAppendPathData(path, sizeof(cmd), cmd, coord);

  /* ストロークの幅を設定 */
  vgSeti(VG_STROKE_LINE_WIDTH, 2);

  /* 指定した位置に描画 */
  vgSeti(VG_MATRIX_MODE, VG_MATRIX_PATH_USER_TO_SURFACE);
  vgLoadIdentity();
  vgTranslate(x, y);

  /* VGPathを描画 */
  vgDrawPath(path, paintModes);

  /* VGPathオブジェクトを破棄 */
  vgDestroyPath(path);
}
```

```
void TestDrawPaintCircle(int width, int height)
{
  VGfloat red[4]   = { 1.0f, 0.0f, 0.0f, 1.0f };
  VGfloat green[4] = { 0.0f, 1.0f, 0.0f, 1.0f };
  VGfloat blue[4]  = { 0.0f, 0.0f, 1.0f, 1.0f };
  VGfloat linearGradient[4] = { 50, 50, 150, 150 };
  VGfloat rampStops[] = {
  0.0f, 1.0f, 0.0f, 0.0f, 1.0f,
  0.5f, 0.0f, 1.0f, 0.0f, 1.0f,
  1.0f, 0.0f, 0.0f, 1.0f, 1.0f
   };
   VGPaint strokePaint, fillPaint;

   /* ペイント座標系とユーザ座標系を一致させる */
   vgSeti(VG_MATRIX_MODE, VG_MATRIX_FILL_PAINT_TO_USER);
   vgLoadIdentity();

   /* 塗りつぶし描画用VGPaintオブジェクトの生成 */
   fillPaint = vgCreatePaint();

   /* 生成したペイントを塗りつぶし描画に設定 */
   vgSetPaint(fillPaint, VG_FILL_PATH);

   /* カラーペイントの設定と描画 */
   vgSetParameteri(fillPaint, VG_PAINT_TYPE, VG_PAINT_TYPE_COLOR);

   /* ペイントカラーの指定 */
   vgSetParameterfv(fillPaint, VG_PAINT_COLOR, 4, red);

   /* ① VGPathを描画(塗りつぶし) */
   DrawCircle(0, 0, VG_FILL_PATH);

   /* ストローク描画用VGPaintオブジェクトの生成 */
   strokePaint = vgCreatePaint();
   vgSetPaint(strokePaint, VG_STROKE_PATH);
   vgSetParameteri(strokePaint, VG_PAINT_TYPE, VG_PAINT_TYPE_COLOR);
   vgSetParameterfv(strokePaint, VG_PAINT_COLOR, 4, green);

   /* ② VGPathを描画(ストローク) */
   DrawCircle(170, 0, VG_STROKE_PATH);
```

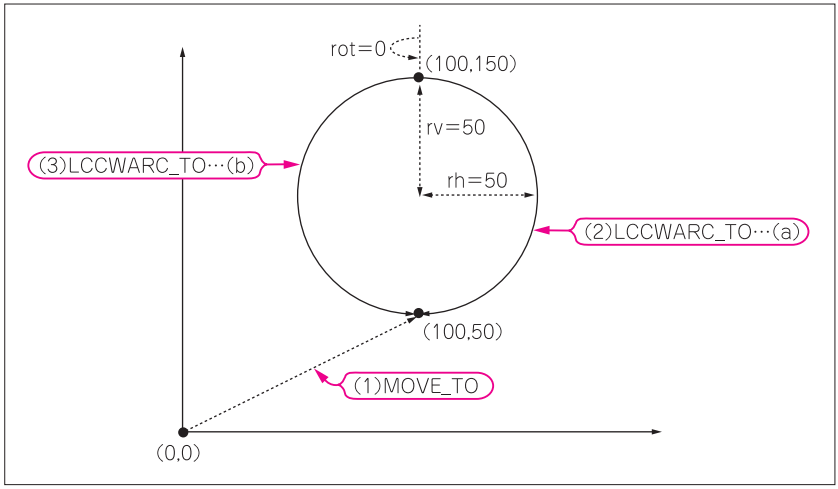
```
  /* ストローク描画用ペイントカラーの再指定 */
  vgSetParameterfv(strokePaint, VG_PAINT_COLOR, 4, blue);

  /* カラーランプテーブル(グラデーション)の設定 */
  vgSetParameterfv(fillPaint, VG_PAINT_COLOR_RAMP_STOPS, 15, rampStops);

  /* 線形グラデーションの設定と描画 */
  vgSetParameteri(fillPaint, VG_PAINT_TYPE, VG_PAINT_TYPE_LINEAR_GRADIENT);

  /* 線形グラデーションのパラメータを設定 */
  vgSetParameterfv(fillPaint, VG_PAINT_LINEAR_GRADIENT, 4, linearGradient);

  /* ③ VGPath を描画(ストローク+塗りつぶし) */
  DrawCircle(340, 0, VG_STROKE_PATH | VG_FILL_PATH);

  /* ペイントオブジェクトの破棄 */
  vgDestroyPaint(fillPaint);
  vgDestroyPaint(strokePaint);
}
```

円形のパスは，楕円弧を描く LCCWARC_TO コマンドを使用して，右半円と左半円に分けて作成します．LCCWARC_TO コマンドのパラメータは，水平半径 rh，垂直半径 rv，楕円弧そのものの回転角 rot(度単位)，終点(x0, y0)の順に指定します．

DrawCircle で描画する円形は，MOVE_TO コマンドで描画の始点を(100, 50)とし，1 回目の LCCWARC_TO コマンドで rh=50，rv=50，rot=0 として半径 50 の右半円を(100, 150)まで描画します．続けて 2 回目の LCCWARC_TO コマンドで同じく rh=50，rv=50，rot=0 として右半円の終点(100, 150)から半径 50 の左半円を(100, 50)まで描画します．結果として，中心の座標(100, 100)，半径 50 の円形のパスが作成されます(**図 4-18**)．

```
VGubyte cmd[] = {
  VG_MOVE_TO_ABS,                          ← 絶対座標指定で始点の移動
  VG_LCCWARC_TO_ABS,                       ← 絶対座標指定で右半円(a)のパスを結ぶ
  VG_LCCWARC_TO_ABS                        ← 絶対座標指定で左半円(b)のパスを結ぶ
};
VGfloat coord[] = {
  100.0f, 50.0f,                           ← 始点の絶対座標を x0, y0 順で指定
  50.0f, 50.0f, 0.0f, 100.0f, 150.0f,      ← 楕円弧を rh, rv, rot, x0, y0 順で指定
  50.0f, 50.0f, 0.0f, 100.0f,  50.0f       ← 楕円弧を rh, rv, rot, x0, y0 順で指定
};
```

図 4-18　DrawCircle で実行するパスの接続過程



次に，ペイント処理の部分を説明します. ①〜③の 3 つの円形に対して，それぞれ異なる修飾を行っています.

① 赤色で円形を塗りつぶす (図 4-17①)

色を付けてパスの描画を行うには，vgCreatePaint で生成した VGPaint オブジェクトが必要です. 生成した VGPaint オブジェクトを塗りつぶし描画専用に適用するには, vgSetPaint の第 2 引数に VG_FILL_PATH を指定して設定します.

なお，(VG_FILL_PATH ¦ VG_STROKE_PATH)を指定すると，指定の VGPaint オブジェクトは塗りつぶし描画とストローク描画の両方に適用されるものになります.

塗りつぶしの方法は 4 種類から選択できます. 単一色(VG_PAINT_TYPE_COLOR)，線形グラデーション（VG_PAINT_TYPE_LINEAR_GRADIENT）と円形グラデーション（VG_PAINT_TYPE_RADIAL_GRADIENT）の 2 種類のグラデーション，パターンペイント（VG_PAINT_TYPE_PATTERN）があります. VG_PAINT_TYPE を指定して vgSetParameteri で設定します.

ここでは, 単一色で塗りつぶすため，VG_PAINT_TYPE_COLOR を設定します.

塗りつぶしの色は, VG_PAINT_COLOR を指定して vgSetParameterfv で設定します. 4 要素の浮動小数点の配列を作成し, R 値, G 値, B 値, アルファ値の順に格納します. それぞれ 0.0 から 1.0 の値で指定します.

R 値, G 値, B 値の 1.0 は最大輝度を意味し, アルファ値の 1.0 は不透過を意味します. 初期値は, R=G=B=0，アルファ値は 1 の不透過の黒です. VGPaint オブジェクトを生成しないで描画を行うときの色も同じ不透過の黒です.

これ以降，VG_FILL_PATH が指定された vgDrawPath を実行すると，パスの内側が指定した色で塗りつぶされます．再度 VG_PAINT_COLOR を指定して vgSetParameterfv で設定し直すと，以降の塗りつぶしは設定し直した色で行われます．

```
VGfloat red[4] = { 1.0f, 0.0f, 0.0f, 1.0f };
VGPaint fillPaint;

fillPaint = vgCreatePaint();
vgSetPaint(fillPaint, VG_FILL_PATH);
vgSetParameteri(fillPaint, VG_PAINT_TYPE, VG_PAINT_TYPE_COLOR);
vgSetParameterfv(fillPaint, VG_PAINT_COLOR, 4, red);
```

②緑色で円形の縁を描画する（図 4-17②）

塗りつぶし描画とストローク描画に分けて色付けする場合は，塗りつぶし描画用とストローク描画用それぞれの VGPaint オブジェクトが必要です．

VGPaint オブジェクトをストローク描画専用に適用するには，vgSetPaint の第 2 引数に VG_STROKE_PATH を指定して設定します．

ここでは，単一色でストローク描画するため，先と同様に vgSetParameteri で VG_PAINT_TYPE パラメータに VG_PAINT_TYPE_COLOR を設定します．

ストローク描画に適用する色は，塗りつぶし描画と同様に，VG_PAINT_COLOR を指定して vgSetParameterfv で設定します．

```
VGfloat green[4] = { 0.0f, 1.0f, 0.0f, 1.0f };
VGPaint strokePaint;

strokePaint = vgCreatePaint();
vgSetPaint(strokePaint, VG_STROKE_PATH);
vgSetParameteri(strokePaint, VG_PAINT_TYPE, VG_PAINT_TYPE_COLOR);
vgSetParameterfv(strokePaint, VG_PAINT_COLOR, 4, green);
```

③青色で円形を縁取り，内側をグラデーションで塗りつぶす（図 4-17③）

図 4-17 の③の円は（VG_FILL_PATH | VG_STROKE_PATH）を指定して vgDrawPath を実行しているので，塗りつぶし描画とストローク描画の両方が実行されます．

ストローク描画は，ストローク描画用の VGPaint オブジェクト strokePaint が適用され，指定の青色でストローク描画されます．塗りつぶし用の VGPaint オブジェクト fillPaint は，再利用して単一色から線形グラデーションに変更して適用します．線形グラデーションで塗りつぶし描画するため，vgSetParameteri で VG_PAINT_TYPE パラメータに VG_PAINT_TYPE_LINEAR_GRADIENT を設定します．

線形グラデーションと円形グラデーションを使用する場合は，カラーランプテーブルを設定します．色の始点の位置を 0.0，色の終点の位置を 1.0 として，合計 32 個の色の分岐点が指定できます．1 つの分岐点の情報は，カラーランプテーブル内の位置，R 値，G 値，B 値，アルファ値の順に 5 つの情報を組にして格納します．

さらに，線形グラデーションを行う場合には，グラデーションを適用する座標を VG_PAINT_LINEAR_GRADIENT を指定して vgSetParameterfv で設定します．始点と終点をペイント座標で指定します．

TestDrawPaintCircle では，始点(50, 50)，終点(150, 150)としていますので、ペイント座標に**図 4-19** の左側のようなグラデーションの領域を生成することになります．一方，円形のパスはユーザ座標で，中心の座標(100, 100)，半径 50 の円形のパスを作成しています．

ペイント座標系は，そのマトリクスを単位行列に初期化していますので，ユーザ座標系と一致します．ユーザ座標の点(x, y)の色は，ペイント座標の点(x, y)の色となります．結果的に円形のパスは，**図 4-19** の右側のようにグラデーションで塗りつぶされます．

その後，このユーザ座標系は描画サーフェス座標系に変換されます．vgTranslate で描画先の原点を(340, 0)に移動しているので，結果として**図 4-17** の③の位置に描画されます．

```
VGfloat rampStops[] = {
  0.0f, 1.0f, 0.0f, 0.0f, 1.0f,      ← 始点の位置と色(赤)とアルファ値(不透過)
  0.5f, 0.0f, 1.0f, 0.0f, 1.0f,      ← 分岐点の位置と色(緑)とアルファ値(不透過)
  1.0f, 0.0f, 0.0f, 1.0f, 1.0f       ← 終点の位置と色(青)とアルファ値(不透過)
};
VGfloat linearGradient[4] = {
    50.0f,  50.0f,                   ← グラデーションの開始位置を x0, y0 順で指定
   150.0f, 150.0f                    ← グラデーションの終了位置を x1, y1 順で指定
};
vgSetParameterfv(fillPaint, VG_PAINT_COLOR_RAMP_STOPS, 15, rampStops);
vgSetParameteri(fillPaint, VG_PAINT_TYPE, VG_PAINT_TYPE_LINEAR_GRADIENT);
vgSetParameterfv(fillPaint, VG_PAINT_LINEAR_GRADIENT, 4, linearGradient);
```

線形グラデーションでは，始点と終点で指定した領域をカラーランプテーブルの位置 0.0 から 1.0 の色に基づいて色付けを行います．この領域の範囲外の色付けについては，始点と終点を拡張（VG_COLOR_RAMP_SPREAD_PAD），カラーランプテーブルの繰り返し（VG_COLOR_RAMP_SPREAD_REPEAT），カラーランプテーブルの順序を反転して繰り返し（VG_COLOR_RAMP_SPREAD_REFLECT）の 3 方式から選択できます．

VG_PAINT_COLOR_RAMP_SPREAD_MODE を指定して vgSetParameteri で設定します．**図 4-19** の左側は領域外を省略していますが，実際は**図 4-20** のような色付けがされます．

図 4-19　線形グラデーションの塗りつぶし描画の例

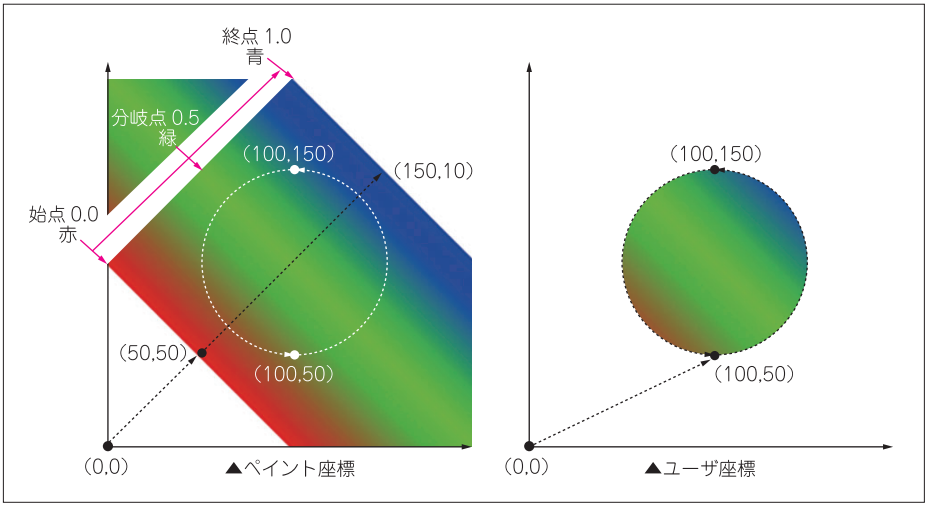
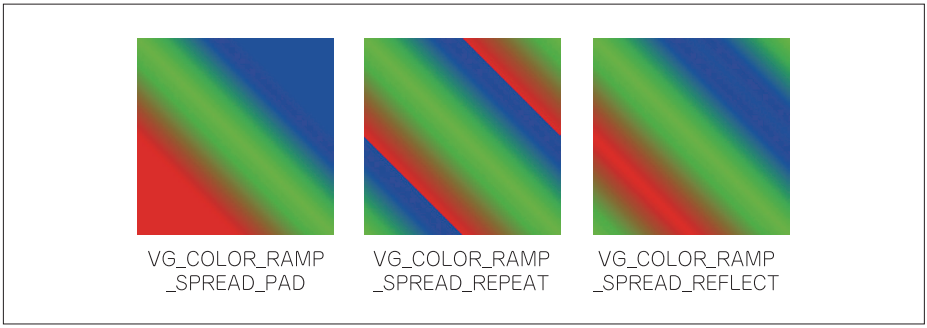


図 4-20　ペイント領域外の塗りつぶしの例



使い終わった VGPaint オブジェクトは, vgDestroyPaint で破棄します.

vgDestroyPaint は, 描画サーフェスの更新(eglSwapBuffers)を待つことなく, vgDrawPath 実行後, その VGPaint オブジェクトを使用することがなければ, すぐに実行することができます.

4-6 OpenVG による背景、画像の事前作成

背景やボタン画像など，よく使用する画像を事前に生成しておくことで，実行時に再描画する負荷を低減できます．

(1) OpenVG による背景の事前生成

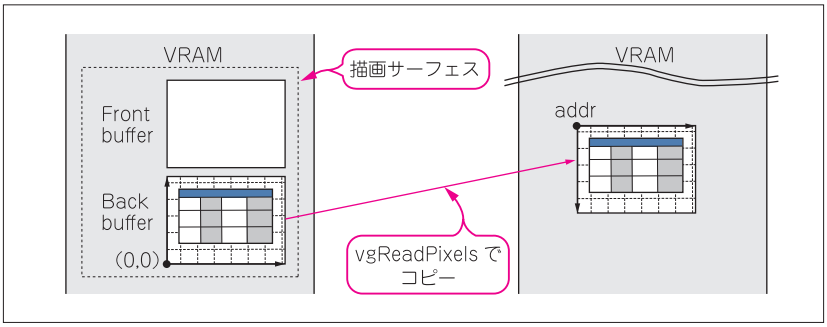
背景を事前に VRAM に作成しておけば，それをウィンドウ合成することで再描画する必要はなくなります．OpenVG で背景を作成するには，描画サーフェスが必要になります．通常使用する描画サーフェスがあれば，そのサーフェスの表示オフ期間中（通常表示前の初期化期間中など）を利用して，事前に作成しておく画像を描画することが可能です．

下記のコードは，OpenVG で描画した後，横 640× 縦 480 画素の画像を ARGB8888 画像としてアドレス addr が示す VRAM 領域に 1 次元コピーします（**図 4-21**）．

コピー完了は vgFinish で待ちます．コピー後は，addr をフレームバッファの先頭アドレスにして DSP でウィンドウ表示することができます．なお，OpenVG と DSP とは座標系が異なるため，Y 軸反転するように引数を設定します．

```
vgReadPixels((void*)((VGuint)addr + 640 * 4 *(480 - 1)), -640 * 4,
             VG_sARGB_8888, 0, 0, 640, 480);
vgFinish();
```

図 4-21　OpenVG による背景の事前作成の例



通常使用する描画サーフェスより背景のサイズが大きい場合，または通常動作中も裏で背景を変更したい場合，EGL でオフスクリーン用の描画サーフェスを構築して利用することができます．このサーフェスは通常の描画サーフェスの構築と同様の操作で，eglCreatePbufferSurface で構築します．

描画前後に eglMakeCurrent で通常使用する描画サーフェスのコンテキストとの切り替えを行う必要があります．

(2) OpenVG による画像の事前作成

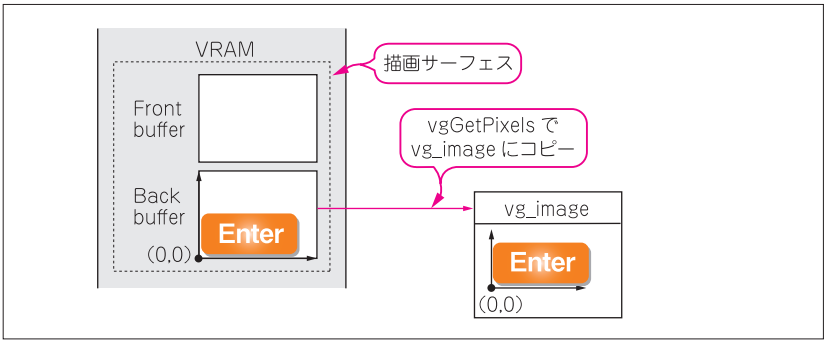
ボタンなどの画像を VGImage オブジェクトとして事前に作成しておけば，描画はコピー操作（vgDrawImage）だけで実現できます．

下記のコードは，vgCreateImage で横 200× 縦 100 画素の ARGB8888 の VGImage オブジェクト vg_image を生成し，vgGetPixels で描画サーフェスにある座標（0, 0）から横 200× 縦 100 画素の画像を vg_image に格納します（図 4-22）．

この操作は，背景の場合と同じように，描画サーフェスの表示オフ期間中に行うことができます．

```
VGImage vg_image;

vg_image = vgCreateImage(
              VG_sARGB_8888, 200, 100,VG_IMAGE_QUALITY_NONANTIALIASED);
vgGetPixels(vg_image, 0, 0, 0, 0, 200, 100);

vgFinish();
```

図 4-22　OpenVG による画像の事前作成の例



(3) PC 画像の取り込み

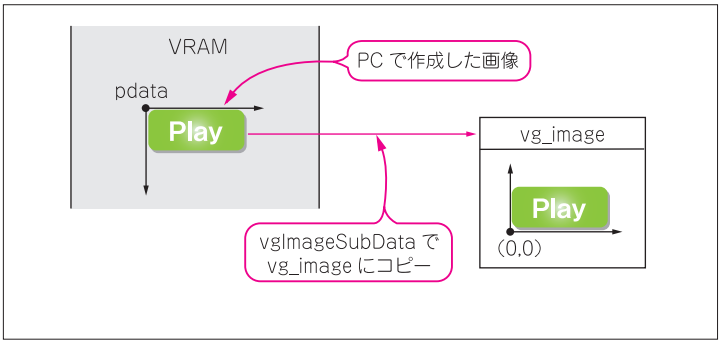
PC などで作成した画像をあらかじめ VRAM に 1 次元格納しておき，VGImage オブジェクトに取り込むことができます．

下記のコードは，PC で作成した横 320× 縦 240 の ARGB8888 のトップダウン画像を vgImageSubData で VGImage オブジェクトに取り込みます(**図 4-23**)．

背景の場合と同じように，座標系が異なる場合は，Y 軸反転して取り込むように引数を設定します．

```
VGImage vg_image;

vg_image = vgCreateImage(VG_sARGB_8888, 320, 240, VG_IMAGE_QUALITY_NONANTIALIASED);
vgImageSubData(vg_image,
               (const void*)((VGuint)pdata + 320 * 4 * (240 - 1)), -320 * 4,
               VG_sARGB_8888, 0, 0, 320, 240);

vgFinish();
```

図 4-23　PC で作成したトップダウン画像の取り込みの例



4-7 OpenVG によるアニメーションの例

vgChildImage により，親画像の一部を参照する VGImage オブジェクトを作成することができます．

下記のコードは，横 968× 縦 88 画素の親画像 vg_image から全 11 フレームの横 88× 縦 88 画素の子画像を vg_anim に割り当てます．所望の周期で vgDrawImage（vg_anim［i］）を実行することでアニメーション表示が行えます（図 4-24）．

```
VGImage vg_anim[11];
int i;

for(i=0; i<11; i++){
  vg_anim[i] = vgChildImage(vg_image, i * 88, 0, 88, 88);
}
```

図 4-24　親画像から子画像の割り当ての例

OpenVG は，レンダリング品質をアンチエイリアス処理なし，高速，最高品質の 3 種類から選んで指定できます．アニメーション中は，アンチエイリアス処理なし，または高速に設定し，アニメーション完了後は最高品質に戻すといった描画時の負荷を最適化できます．

レンダリング品質は，VG_RENDERING_QUALITY を指定して vgSeti で設定します．

VG_RENDERING_QUALITY_NONANTIALIASED の指定は，アンチエイリアス処理を無効にします．VG_RENDERING_QUALITY_FASTER を指定すると，OpenVG の API 適合基準を満たしながら，利用可能な最高速度で描画を行います．VG_RENDERING_QUALITY_BETTER を指定すると，利用可能な最高品質で描画が行われます．

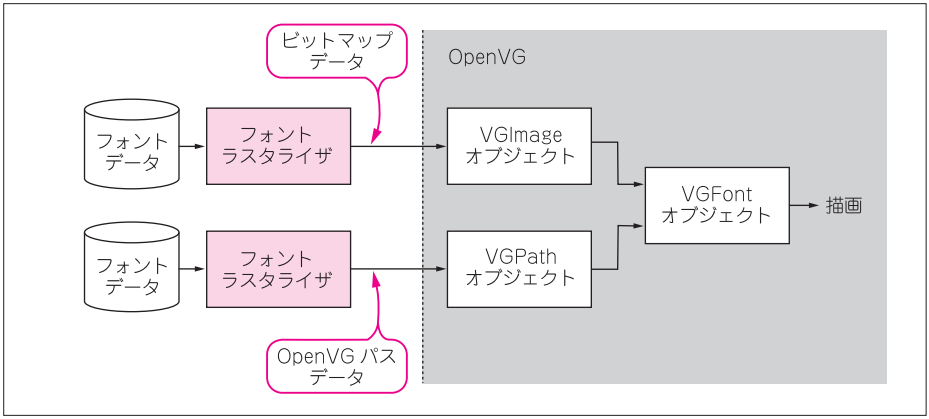
```
vgSeti(VG_RENDERING_QUALITY, VG_RENDERING_QUALITY_NONANTIALIASED);
vgSeti(VG_RENDERING_QUALITY, VG_RENDERING_QUALITY_FASTER);
vgSeti(VG_RENDERING_QUALITY, VG_RENDERING_QUALITY_BETTER);
```

4-8 OpenVG による文字の描画

OpenVG では，スケーラブルフォント（アウトラインフォントなど）とビットマップフォントが扱え，フォントを管理して文字列の描画を容易にする API が規定されています．ただし，フォントデータから実際の文字画像を生成する API は規定されていないため，別途フォントラスタライザと呼ばれるソフトウェアが必要になります（図 4-25）．

例えば，フォントデータからビットマップデータを得るには，FreeType2 などのラスタライザが使用できます．得られたビットマップデータは，vgImageSubData で VGImage オブジェクトに変換できます．フォントデータから OpenVG のパスデータ（VGPath オブジェクト）を得るには，ダイナコムウェア社の DigiType API などが使用できます．

図 4-25　文字の描画にはフォントラスタライザが必要



フォントの管理は，VGFont オブジェクトを作成して管理します．文字データが VGPath オブジェクトの場合は vgSetGlyphToPath を使用して VGFont オブジェクトに登録します（VGImage オブジェクトの場合は vgSetGlyphToImage）．下記に使用例を示します．

```
VGPath vg_path;
for(id=0; id<MAX_TEXT_ID; id++)]{
  vg_path = vgCreatePath(VG_PATH_FORMAT_STANDARD, VG_PATH_DATATYPE_F, 1.0f, 0.0f,
              0, 0, VG_PATH_CAPABILITY_ALL);
  raster_gryph(&text[id], &vg_path);  /* フォントラスタライザでパスデータを取得 */
  vgSetGlyphToPath(vg_font, id, vg_path, VG_FALSE, origin, escapement);
  vgDestroyPath(vg_path);
}
```

※ raster_gryph は OpenVG の API ではありません．

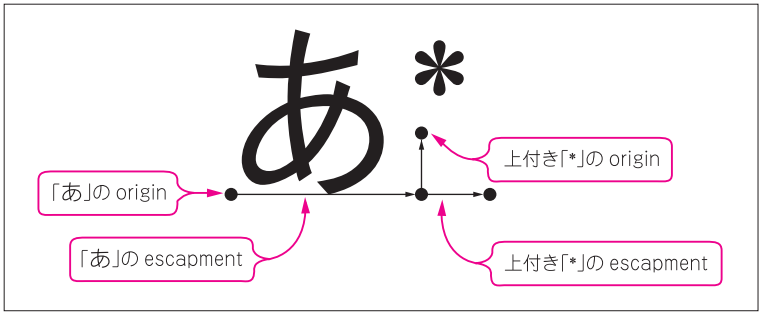
vgSetGlyphToPath に指定する id は，vg_path が 1 文字のパスデータの場合，unicode などの文字コードを指定するのが良いでしょう．vg_path が文字列を示す場合，id はアプリケーション管理の id などを指定します．文字の描画時には，ここで指定した id を指定して描画します．1 つの id で文字列を登録した場合は，その文字列が描画されます．

origin は，文字のパスデータと同じ座標系で，原点の位置を指定します．origin[0]が水平方向，origin[1] が垂直方向の原点位置です．上付き文字など位置調整が必要な場合は，origin を用いて移動量を指定します．通常は origin={ 0, 0 } とします．

escapement は，文字のパスデータと同じ座標系で，次の文字の描画位置までの移動量を指定します．escapement [0] が水平方向，escapement [1] が垂直方向の移動量です．通常は垂直方向の移動量は 0 で，escapement={ 文字幅 , 0 } とします（**図 4-26**）．

VGFont オブジェクトに登録する VGPath オブジェクトは，登録後に vgDestroyPath を実行して VGFont オブジェクトの解放時に合わせて解放するように予約できます．

図 4-26　文字の配置例



文字の描画は，vgDrawGlyph（複数 id の場合は vgDrawGlyphs）を使用して，vgSetGlyphToPath で登録したときの id を指定して描画します．文字の描画位置は OpenVG で管理されているため，次に vgDrawGlyph（または vgDrawGlyphs）を実行した場合は，前の文字（または文字列）に続いて escapement だけ移動した位置に描画されます．

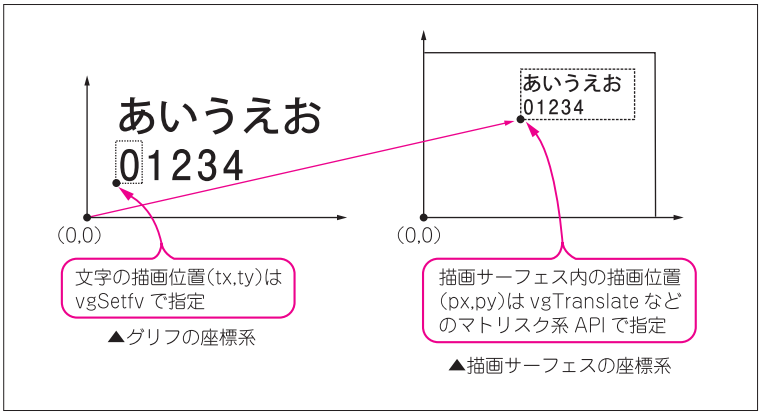
なお，文字の描画開始位置は，vgSetfv で直接指定できます．次の行に移動したい場合は，vgSetfv で水平位置は行頭，垂直方向には行間分だけ下げた位置を指定します．

```
vgSeti(VG_MATRIX_MODE, VG_MATRIX_GLYPH_USER_TO_SURFACE);
vgLoadIdentity();
vgTranslate(px, py);
origin[0] = tx;
origin[1] = ty;
vgSetfv(VG_GLYPH_ORIGIN, 2, origin);
vgDrawGlyph(vg_font, id, VG_FILL_PATH, VG_FALSE);
```

VG_MATRIX_MODE に VG_MATRIX_GLYPH_USER_TO_SURFACE を指定することで，フォント描画用のマトリクスでアフィン変換が行えます．文字領域内（グリフ座標系）では vgSetfv で描画位置を指定し，描画サーフェス内の描画位置は vgTranslate などマトリクス系の API で指定します(**図 4-27**)．

描画位置を描画サーフェス内の原点に移動するには vgLoadIdentity を実行しますが，文字に関しては，さらに vgSetfv で描画位置を origin={ 0, 0 } にクリアする必要があります．

図 4-27　文字描画位置の指定例



4-9 OpenVG による色変換と合成

OpenVG の描画処理の最終段で色変換が行われ，続いて画像間の合成が行われます．

色変換を有効にした場合，（R, G, B, A）の各チャネルは，次式に示すように，スケール係数（−127.0 〜 +127.0）が乗算され，チャネルごとのバイアス値（−1.0 〜 +1.0）が加算されて，（R', G', B', A'）に変換できます．

変換後の各チャネルの値は，0.0 〜 1.0 の範囲にクランプされます．

$$A' = A \times S_a + B_a$$
$$R' = R \times S_r + B_r$$
$$G' = G \times S_g + B_g$$
$$B' = B \times S_b + B_b$$

スケール係数とバイアス値は，8 つの浮動小数点ベクトル（$S_r, S_g, S_b, S_a, B_r, B_g, B_b, B_a$ 順）として設定します．

次のコードの scale 変数の値を 0.0 〜 1.0 の間で変化させることで，描画色や画像を操作することなく，フェードイン，フェードアウトなどの効果を演出することができます．

```
/* Sr, Sg, Sb, Sa, Br, Bg, Bb, Ba */
VGfloat values[ ] = { 1.0f, 1.0f, 1.0f, 1.0f, 0.0f, 0.0f, 0.0f, 0.0f };
values[0] = values[1] = values[2] = scale;
vgSetfv(VG_COLOR_TRANSFORM_VALUES, 8, values);
vgSeti(VG_COLOR_TRANSFORM, VG_TRUE);
```

画像間の合成方法として，**表 4-4** に示す 10 種類のモードが定義されています（初期値は VG_BLEND_SRC_OVER モード）．

これから描画する画素（C_{src}, α_{src}）と，すでに描画済みの画素（C_{dst}, α_{dst}）が与えられると，**表 4-4** に示す色の合成関数 $c'(C_{src}, C_{dst}, \alpha_{src}, \alpha_{dst})$，およびアルファ値の合成関数 $\alpha(\alpha_{scr}, \alpha_{dst})$ に従い，画素が置き換わります．なお，**表 4-4** の合成関数 $c'()$ はアルファ値が乗算された値を示すので，乗算前の値は $c'()/\alpha()$ によって得られます．

表 4-4　OpenVG の画像合成モードと合成関数の関係

VG_BLEND_MODE	$c'(C_{src}, C_{dst}, \alpha_{stc}, \alpha_{dst})$	$\alpha'(\alpha_{src}, \alpha_{dst})$
VG_BLEND_SRC	$\alpha_{src} * C_{src}$	α_{src}
VG_BLEND_SRC_OVER	$\alpha_{src} * C_{src} + \alpha_{dst} * C_{dst} * (1 - \alpha_{src})$	$\alpha_{src} + \alpha_{dst} * (1 - \alpha_{src})$
VG_BLEND_DST_OVER	$\alpha_{src} * C_{src} * (1 - \alpha_{dst}) + \alpha_{dst} * C_{dst}$	$\alpha_{src} * (1 - \alpha_{dst}) + \alpha_{dst}$
VG_BLEND_SRC_IN	$\alpha_{src} * C_{src} * \alpha_{dst}$	$\alpha_{src} * \alpha_{dst}$
VG_BLEND_DST_IN	$\alpha_{dst} * C_{dst} * \alpha_{src}$	$\alpha_{dst} * C_{src}$
VG_BLEND_MULTIPLY	$\alpha_{src} * C_{src} * (1 - \alpha_{dst}) + \alpha_{dst} * C_{dst} * (1 - \alpha_{src}) + \alpha_{src} * C_{src} * \alpha_{dst} * C_{dst} *$	$\alpha_{src} + \alpha_{dst} * (1 - \alpha_{src})$
VG_BLEND_SCREEN	$\alpha_{src} * C_{src} + \alpha_{dst} * C_{dst} - \alpha_{src} * C_{src} * \alpha_{dst} * C_{dst}$	$\alpha_{src} + \alpha_{dst} * (1 - \alpha_{src})$
VG_BLEND_DARKEN	$min(\alpha_{src} * C_{src} + \alpha_{dst} * C_{dst} * 1 - \alpha_{src}), \alpha_{src} * C_{src} * (1 - \alpha_{dst}) + \alpha_{dst} * C_{dst})$	$\alpha_{src} + \alpha_{dst} * (1 - \alpha_{src})$
VG_BLEND_LIGHTEN	$max(\alpha_{src} * C_{src} + \alpha_{dst} * C_{dst} * (1 - \alpha_{src}), \alpha_{src} * C_{src} * (1 - \alpha_{dst}) + \alpha_{dst} * C_{dst})$	$\alpha_{src} + \alpha_{dst} * (1 - \alpha_{src})$
VG_BLEND_ADDITIVE	$min\left(\frac{\alpha_{src} * C_{src} + \alpha_{dst} * C_{dst}}{min(\alpha_{src} + \alpha_{dst}, 1)}, 1\right)$	$min(\alpha_{src} + \alpha_{dst}, 1)$

C_{src}はアルファ値が乗算されていない RGB 値で，各 R_{src}，またはC_{src}，または B_{src} を示します. また，C_{dst} はアルファ値が乗算されていない RGB 値で，各 R_{dst} ，または G_{dst} ，または B_{dst}を示します.

表示回路で実現したクロスフェードの演出は，OpenVG の色変換と合成処理でも実現することができます. 最初に遷移後の新しい画像を VG_BLEND_SRC モードで描画し，次に VG_BLEND_SRC_OVER モードに変更して, その画像に重ねるように遷移前の現在の画像を上書きします.

その際, 現在の画像は, 色変換で $S_r = S_g = S_b = 1.0$, $B_r = B_g = B_b = B_a = 0.0$とし，$S_a$に所望のアルファ値 0.0（0%）～ 1.0（100%）の範囲で設定します. 色変換により，アルファ値が 1.0 の画像でもアルファ値を操作できます. 結果として，$S_a * C_{src} + C_{dst} * (1 - S_a)$に従った S_aで合成された画像が得られます.

画像合成モードは, VG_BLEND_MODE を指定して, vgSeti で設定します.

AG903 のブロック図, 外観, 主な仕様を以下に示します.

図 A　AG903 のブロック図

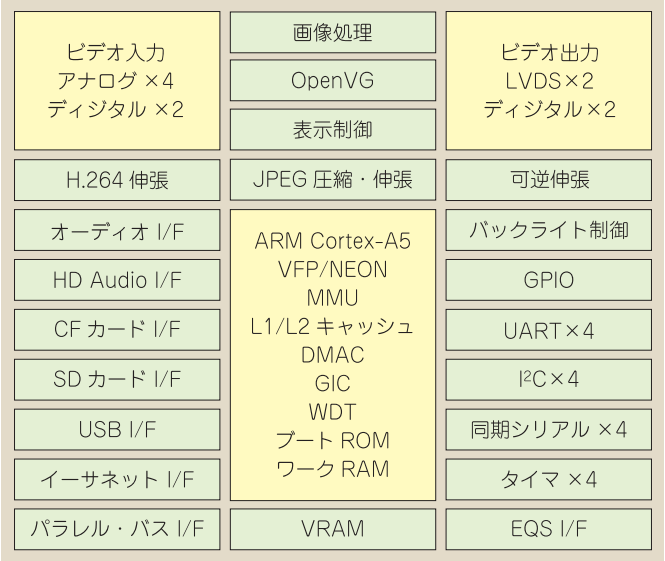


写真 1　AG903 の外観
パッケージは QFP256 ピン, サイズは 28×28mm

Column　AG903 内蔵 GPU による AI 処理の高速化！

ax 株式会社と株式会社アクセルが開発した組み込み向けのディープラーニング推論エンジン「ailia SDK for RTOS」により, AG903 で AI 処理が可能です. NEON を使用した場合の国旗カードの識別時間は 195msec でしたが, AG903 内蔵 GPU を利用することで, 同識別時間は 28msec と大幅に高速化されました.

図 B　AG903 による国旗識別デモ

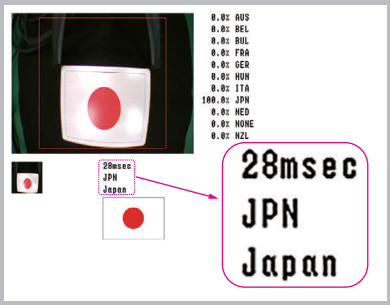


図 C
ailia SDK for RTOS の実行フロー, 推論エンジンの生成に標準的な AI フレームワークを利用できる.

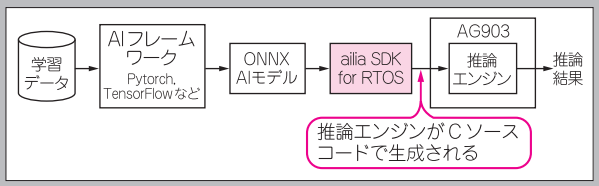


表 A　AG903 の主な仕様

機 能	詳 細	性 能
内蔵 CPU	コア, クロック	ARM Cortex-A5, 400MHz
	FPU	ARM VFP3/NEON
	周辺機能	MMU, L1/L2 キャッシュ, DMAC, GIC, タイマ
内蔵 VRAM	容量	1G ビット(128M バイト)
ビデオ入力	入力 I/F	24 ビット・デジタル ×1 または 8 ビット・ デジタル ×2, アナログ ×4
	入力フォーマット	RGB888/RGB565/YCbCr422(デジタル) NTSC/PAL コンポジット(アナログ)
	最大解像度	4096×4096 ドット (演算パイプライン 1920 ドット ×1, 640 ドット ×1)
	入力ドット・クロック	最大 170MHz
ビデオ出力	出力 I/F	24 ビット・デジタル ×1 または 8 ビット・デジタル ×2, LVDS デュアル ×1 または LVDS シングル ×2
	出力フォーマット	RGB888/RGB565/YCbCr422(デジタル)
	最大解像度	1920×1200, 60Hz(デジタル) 1366×　768, 60Hz(LVDS シングル) 1920×1200, 60Hz(LVDS デュアル)
	出力ドット・クロック	最大 170MHz
	発色数	最大 24 ビット
	機能	16 ウィンドウ(2 画面合計), 拡大 / 縮小 , 反転 , 色補正 ウィンドウ間合成 , 色空間変換 , ディザリング , バックライト制御
描画	対応 API	OpenVG1.1 準拠 API, AG9 描画 API
画像圧縮 / 伸張	圧縮方式	JPEG(圧縮 / 伸張), H.264(伸張), AG9 形式(可逆伸張)
画像処理	機能	I/P 変換, クロップ, ノイズフィルタ, 色空間変換, HSV/HLS 変換, 縮小, 空間フィルタ, 濃度変換, 閾値処理, ラベリング, ヒストグラム生成, フレーム間演算, ディザリング
外部 I/F	CPU 周辺 I/F	UART×4,　同期シリアル ×4,　I2C×2, タイマ ×4, バックライト制御 ×2,　EQS×1, GPIO,　デバッガ
	イーサネット	MII/RMII(10/100M)×1
	USB	USB2.0(HS/FS/LS)×1 (ホスト / ファンクション切替)
	オーディオ	I2S/ 右詰め / 左詰め /TDM×4, HD Audio×1
	メディア	CF カード(TrueIDE/PC カード), SD カード(SDSC/SDHC/SDIO/MMC)
パラレル・バス I/F	バス幅, クロック	32/16/8 ビット, 最大 66MHz
	アドレス空間	26 ビット(64M バイト)
	メモリ I/F	SRAM/SDRAM I/F
デバイス・モード	バス幅, クロック	32/16 ビット, 最大 66MHz
	アドレス空間	26 ビット(64M バイト)
	データ転送モード	デュアル・アドレス・モード
クロック		24/25/27/30/48/50MHz
動作電圧		コア : 1.15V,　I/O : 1.8V または 3.3V,　アナログ : 1.8V, VRAM : 1.8V
パッケージ		256 ピン QFP(底面放熱パッド付き), 28×28mm

著者略歴

●麻生　勝之(あそう・かつゆき)
株式会社 アクセル. スキーとお酒をこよなく愛するエンジニア.
ジョギング中に「やり切るまで続けないのはいけない」と考える毎週末.

●宮崎　仁(みやざき・ひとし)
有限会社 宮崎技術研究所. 一人で何役もこなすユーティリティ・エンジニアを目指すも, なかなか道はけわしいと思う今日この頃.

トランジスタ技術 特別小冊子(非売品)

いまさら聞けない組み込みGUI!!

著　者　麻生 勝之 / 宮崎 仁
発行人　小澤 拓治
編集人　中元 正夫
発行所　CQ出版株式会社
〒112-8619　東京都文京区千石4-29-14
(03)5395-2123(出版部)
(03)5395-2141(販売部)
振替　00100-7-10665
Printed in Japan

デザイン / DTP(株)イー・スリー・プロダクション

わきまえた、煌めき。

京都室町に280年続く老舗の帯匠・誉田屋の十代目山口源兵衛。原始布や古代繭、天蚕といった素材にこだわり、類い希な技術で見事な作品を完成させることで、古から綿々と続く伝統を受け継ぎながら、その先に繋がる未来へと、帯の新境地を開拓し続けています。山口源兵衛の帯は、丁寧な一つ一つの駒さばきが、帯全体を構成するための強い意志に裏打ちされ、隅々まで入念に仕上げられた芸術としての雰囲気を醸し出しています。

その姿は比類のない精緻さや完成度を誇りながら、徒に自己を主張するのではなく、着物と組み合わせた和装全体での調和が考慮された、非常に穏やかな表情を見せています。そこには一分の隙も無駄も存在していません。

己の役割をわきまえたもののみがもつ端正な佇まいと美しさ、その輝きは主役たる着物に決して引けを取るものではありません。

私たちもこの帯のようなSoCを目指しています。

組み込み機器向けグラフィックスLSI

AG903

AG903 主な仕様

描画エンジン	OpenVG™ 1.1 対応
CPUコア	Arm®Cortex®-A5 400MHz
内蔵DRAM容量	64Mバイト（512Mbit）／128Mバイト（1Gbit）
ビデオ入力	CMOSデジタル（24bit）×1、アナログ（NTSC/PAL）×4
ビデオ出力	CMOSデジタル（24bit）×1、LVDSデュアル×1
最大解像度	1920×1200（1出力時）
画像コーデック	JPEG（圧縮伸長）、H.264（伸長）
電源電圧	コア1.15V、I/O1.8または3.3V（選択可能）、アナログ1.8V、DRAM1.8V
パッケージ	QFP256pin、28mm×28mm
その他	画像処理機能、外部CPUからの制御

※ArmおよびCortexはArm Ltd.またはその子会社の登録商標です。
※OpenVGはThe Khronos Group Inc.の商標です。
※その他の社名、製品名などは、一般に、各社の商標または登録商標です。

AXELL CORPORATION
株式会社アクセル
〒101-8973 東京都千代田区外神田4-14-1 秋葉原UDX南ウイング10階
TEL 03-5298-1670 FAX 03-5298-1671 EMAIL: sales@axell.co.jp
詳しくはこちらへ
https://www.axell.co.jp/