Autodesk®
Scaleform®

HUD キットの概要

このドキュメントでは、AAA クオリティのファーストパーソンシューティングゲームに求められるユーザーインタフェースの開発を支援する、再利用可能かつ豊富な機能を備えた Scaleform 4.3 HUD キットについて説明しています。とくに、ユーザーインタフェースの管理に使われる Flash UI コンテンツと C++コードについて焦点を当てています。

作者　Nate Mitchell、Prasad Silva

バージョン　2.0

最終更新日　2010 年 7 月 30 日

Autodesk®
GAMEWARE

Copyright Notice

Autodesk® Scaleform® 4.3

EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE REGARDING THESE MATERIALS.

Autodesk Scaleform の連絡先：

ドキュメント名	HUD キットの概要
住所	Autodesk Scaleform Corporation 6305 Ivy Lane, Suite 310 Greenbelt, MD 20770, USA
ウェブサイト	www.scaleform.com
Email	info@scaleform.com
電話	+1 (301) 446-3200
Fax	+1 (301) 446-3199

目次

1 はじめに

Scaleform の HUD（ヘッドアップディスプレイ）キットは、高性能、フル機能、AAA クオリティのユーザーインタフェースキットシリーズの最初の開発環境で、ユーザーインタフェースのカスタマイズや開発したゲームへの落とし込みを支援するツールです。新たにサポートされた Scaleform® Direct Access API を活用することで、高性能なファーストパーソンシューティング UI の作成が可能です。

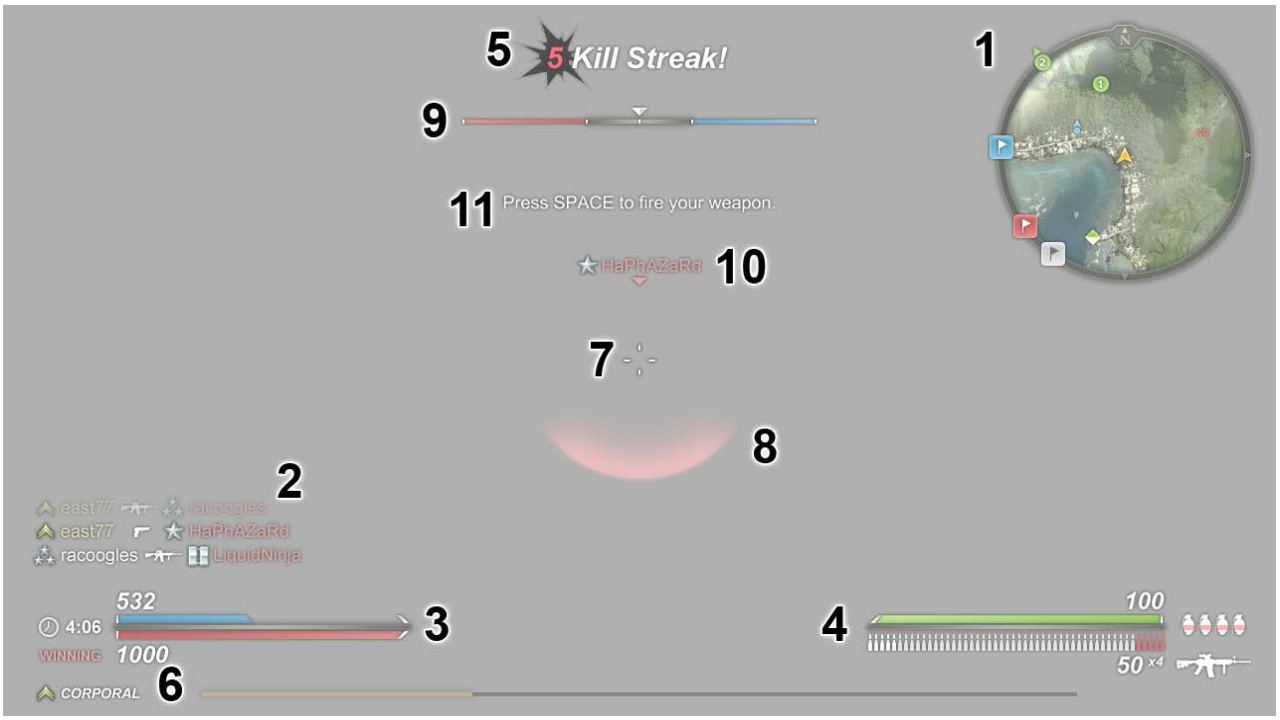

図 1: HUD の概観

HUD キットは再利用可能な以下の UI エレメントに対応しています。

1. ゲーム minimap
2. ゲームイベントのアニメーション付きログ
3. 得点とチーム統計情報
4. プレイヤーのヘルス／銃弾数／ステータス
5. アニメーション付きポップアップ表示
6. 経験値とランク表示
7. ダイナミックレティクル
8. 方向性ヒットインジケータ
9. 目標物キャプチャインジケータ
10. ビルボード／名前
11. プレイヤーテキスト表示

このキットは FPS ゲーム用の UI ソリューションを短期間で開発するものですが、キットの内容によってユーザーが制限されることはありません。Scaleform は、ユーザーの皆様が本キットの各エレメントをカスタマイズあるいは拡張を行うことで、さまざまなゲームやアプリケーションを実現する革新的なインタフェースを創出していただきたいと考えています。

2 概要

2.1 ファイルの位置とビルドの注記

このデモに関連するファイルは以下のディレクトリに格納されています。

- *Apps\Kits\HUD* - HUD デモ用の実行可能な C++コードが格納されています。
- *Bin\Data\AS2 or AS3\Kits\HUD* - Flash 素材と ActionScript コードが格納されています。(AS2 ディレクトリは ActionScript2/Flash8 アセットを含みます。AS3 ディレクトリは ActionScript3/Flash10
  アセットを含みます)
- *Projects\Win32 \{Msvc80* または *Msvc90}\Kits\HUD* – Microsoft Windows 上で動作する Visual Studio 2005/2008 用のプロジェクトが格納されています。
- *Projects\Xbox360\{Msvc80* または *Msvc90}\Kits\HUD* – Xbox 360 上で動作する Visual Studio 2005/2008 用のプロジェクトが格納されています。
- *Projects\Common\HudKit*.mk – HUD キット用の PS3 makefile です。

Windows 用デモのビルト済み実行可能ファイルである HUDKit.exe は *Bin\Kits\HUD* に格納されています。このファイルは Scaleform SDK ブラウザのスタートメニューからもアクセスが可能です。

Windows の場合、本デモのビルドおよび実行には、*Projects\Win32\Msvc80\Kits*（または *Msvc90\Kits*）ディレクトリに格納されている Scaleform 4.3 Kits.sln ファイルを使用します。本ソリューションからデモを実行する前に、デバッグ用の「ワーキングディレクトリ」が *Bin\Data\AS2\Kits\HUD or Bin\Data\AS3\Kits\HUD* に設定されていることを確認してください。

Xbox 360 の場合、デモのビルド、展開、および実行には、*Projects\Xbox360\{Msvc80* または *Msvc90}\Kits\HUD* ディレクトリに格納されている Scaleform 4.3 Kits.sln ファイルを使用します。*Bin\Data\AS2 or AS3\Kits\HUD* に格納されているデモ用のすべての素材は、コンパイル完了後に、対象の Xbox360 に展開されます。

PS3 の場合、make コマンドは Scaleform のインストールディレクトリのルートから実行してください。このコマンドは、デフォルトで、HUD キットに含まれるすべての利用可能なデモをビルドします。PS3 では実行可能ファイルは SN システムツールセットを介して起動します。実行可能ファイルは、*Bin\PS3* にビルドされ、次のオプションを使って起動しなければなりません。

- app_home/ Directory: *{Local Scaleform Directory}\Bin\Data\AS2 or AS3\Kits\HUD*

2.2 デモの使い方

HUD キットの使い方を分かりやすく説明するために、戦場に広がる目標物のコントロールを巡って赤と青の 2 つのチームがバトルを繰り広げるという、きわめて単純なゲームシミュレーションのプロジェクトを用意しました。

目標物のキャプチャ（minimap 上にマーク）と敵プレーヤーの排除によって得点が加算されます。キャプチャした目標物をコントロールしているチームには 3 秒おきに 1 ポイントが与えられます。敵を殺すと 1 ポイントが与えられます。500 ポイントを先に得たチーム、またはラウンド時間（15 分）で最高スコアを得たチームを、勝利者として宣言します。

ゲーム開始時は、装備として、マシンガン×1 丁、ピストル×1 丁、グレネード（手榴弾）×4 発が与えられ、銃弾はフルに装填された状態です。プレーヤーが殺されると、そのプレーヤーの武器と銃弾は再装備されます。

回転する緑色のダイヤモンドで示されるパワーアップが戦場のランダムな場所に発生します。パワーアップを拾うと、プレイヤーのヘルスとランダムに選ばれるいずれかの武器の銃弾数に、あらかじめ決められたブーストが与えられます。

2.3 制御方法

2.3.1キーボード（Windows）

W、A、S、D	ユーザープレーヤーを制御します。W と S はプレーヤーを前方または後方にそれぞれ移動します。A と D はプレーヤーを左または右にそれぞれ移動します。
スペース	現在の武器を発射します。範囲、ダメージ、連射率は、装備している武器それぞれで異なります。
R	現在の武器に装弾します。
G	グレネードを投げます。範囲内のプレーヤーのもっとも近い相手がターゲットとなり爆発し、爆風の範囲にいるすべての敵に損傷を与えます。
ESC	アプリケーション統計欄の表示を、タイトルバーとテキストフィールドとでトグルで切り替えます
B	ユーザーのプレーヤーの AI 制御をトグルします。解除するまでゲームプレイ入力は無効になります。
1、2	武器を変更します。 • 1 でピストルを選択します。 • 2 でライフルを選択します。

2.3.2コントローラ（Xbox/PS3）

D パッド **左ジョイスティック**	ユーザープレーヤーを制御します。W と S はプレーヤーを前方または後方にそれぞれ移動します。A と D はプレーヤーを左または右にそれぞれ移動します。
右トリガー	現在の武器を発射します。範囲、ダメージ、連射率は、装備している武器それぞれで異なります。
X / □	現在の武器に装弾します。
左トリガー	グレネードを投げます。範囲内のプレーヤーのもっとも近い相手がターゲットとなり爆発し、爆風の範囲にいるすべての敵に損傷を与えます。
B / ○	アプリケーション統計欄の表示を、タイトルバーとテキストフィールドとでトグルで切り替えます
Y / △	武器を順に切り替えます。

3 アーキテクチャ

HUD キットは、Flash UI 素材と、HUD のビジュアルステートを更新する C++コードとで構成されています。また、ダミーデータを使った HUD の駆動環境を実現する、C++シミュレーションも備えます。

3.1 C++

HUD ビューのインタフェースと、ビューにデータを与える環境のふたつが、C++コードの主要な部分です。MVC パラダイムでいえば、インタフェースはコントローラに、環境はモデルに、Flash UI はビューに相当するでしょう。HUD キットのビューのコアとなるインタフェースは、Fx をプレフィクスとする各ファイルおよび各クラスによって定義されます。デモまたはゲームシミュレーションのインタフェースは、このプレフィックスのないファイルによって定義されます。

HUD ビューからシミュレーションを分離する目的で、アダプタデザインパターンが実装されています。アダプタはシミュレーションと HUD ビュー間のリクエストを翻訳します。新しいゲームの HUD ビューにインタフェースするには、ユーザーのゲームと HUD ビューとをインタフェースする専用のアダプタクラスを開発しなければなりません。すなわち開発者は、開発したゲームのコードを大幅に変更することなく、HUD キットを再利用できることを意味します。

このドキュメントで詳細が記載されているコードでは、新しい Scaleform Direct Access API を活用しながらこれまでにない高速な UI アップデートとアニメーションを実現するという、Scaleform の最適な実装方法を示しています。

C++コードは次のファイルから構成されています（*Apps\Kits\HUD*とそのサブフォルダに格納されています）。

3.1.1デモ

- **HUDKitDemo.cpp** – スタンダード Scaleform Player の最上位にビルドされたコアアプリケーション。シミュレーションの開始、メンバーの初期化、C++コントローラを使った SWF ファイルのローディングおよび登録を処理します。
- **HUDKitDemoConfig**. – FxPlayerConfig.h に基づく HUDKitDemo 用コンフィギュレーション。コントローラマッピングと Windows アプリケーション定義を含みます。

3.1.2HUD/Minimap ビューコア

- **FxHUDKit.h** – HUD ビューの更新に使うコア HUD キットタイプ。
- **FxHUDView.h/.cpp** – ログやビルボードシステムなど、HUD ビューが使用するタイプを与えます。
- **FxMinimap.h** – ゲーム環境およびアプリケーションが minimap ビュー上に実装するインタフェースを宣言します
- **FxMinimapView.h/.cpp** – minimap ビューで使われるタイプを与えます。

3.1.3ゲームシミュレーション

- **HUDAdapter.h/.cpp** – HUD からシミュレーションを分離して再利用性を向上させる、アダプタデザインパターンの実装。
- **HUDEntityController.h/.cpp** – シミュレーションロジックと AI ビヘイビアの更新に主に使われるエンティティの、コントローラタイプを宣言します。
- **HUDSimulation.h/.cpp** – ゲーム UI の駆動に必要なデモ環境を実装するタイプを与えます。この環境には、制圧防衛（キャプチャ＆ホールド）タイプのゲームで使われる 2 チーム分のプレーヤーが含まれます。

3.2 Flash

HUD キットの Flash コンテンツは **HUDKit.fla** と **Minimap.fla** の 2 つのファイルに分割されています。両方の FLA ファイルは、Scaleform を使った操作に必要な HUD キットの UI エレメントを対象に、定義と配置を行います。また、両方の FLA ファイルには、UI に表示されるすべてのイメージとアイコンが含まれています。

両方のファイルともに複数のレイヤーに分割されています。一般に、各コンポーネント、または類似コンポーネントで構成される各グループには、専用レイヤーが割り当てられます。レイヤー構造は、深さ方向に整列しているエレメントのオーサータイムコントロールに有用です。最上位レイヤーはそのほかのレイヤーの上位に表示され、その他のレイヤーについても同様です。

Scaleform の場合、Flash アニメーションは、C++、ActionScript、または Flash タイムラインを使って扱われます。このデモでは、ほとんどの HUD アニメーションは、Classic Tweens を介した Flash タイムライン上で扱われます。これらアニメーションは一般に、C++からの GFx::Value::GotoAndPlay()コールでトリガーされます。

3.2.1HUDKit.fla

Bin\Data\AS2 or AS3\Kits\HUD ディレクトリにある HUDKit.fla ファイルは、このデモのプライマリ SWF です。このファイルは HUDKitDemo によってランタイムにロードされます。ランタイム時に HUDKit.fla によってロードされる minimap ビューを除いて、すべての Flash UI エレメントはこのファイル内に存在します。

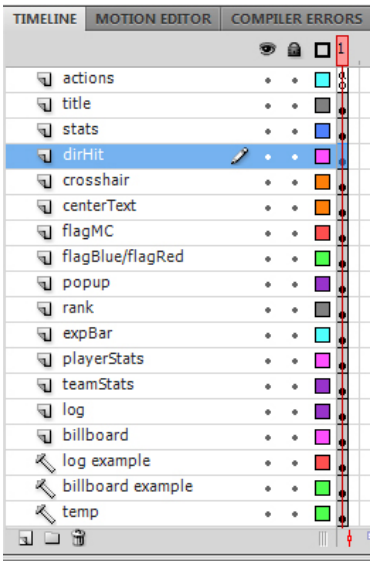

図 2: HUDKit.fla のレイヤー

HUDKitDemo は、タイムラインの Frame 1 で、External Interface を使って HUDKit MovieClip を登録します。

```
ExternalInterface.call("registerHUDView", this);
```

この処理によって、ユーザーインタフェースを操作するために C++が C++ HUDView に登録するポインタが、MovieClip に渡されます。

各レイヤー、MovieClip、レイアウト、アニメーション、および Scaleform C++マネージャについては、本ドキュメントの HUD View セクションで詳しく説明します。

3.2.2Minimap.fla

図 3: Minimap のシンボル

Minimap.fla は Minimap のコアである「Minimap」シンボルで構成されます。なお、デモの目的から考えて、バックグラウンドはインタラクティブな 3D 環境ではなく静的画像で作成しています。

プレーヤーは中央に黄色い矢印として表され、その頭が向いている方角は minimap の境界円周上に示されます（北方向の表示あり）。プレーヤーのほか、minimap ビューには 5 種類のアイコンタイプが表示されます。

- 味方プレーヤー（青）。方向矢印は高ズームのときのみ現れる
- 敵プレーヤー（赤）。方向矢印は高ズームのときのみ現れる
- キャプチャポイント（旗で表示。白は中立、青は味方、赤は敵）
- パワーアップ（回転する緑と白のアイコン）

キャプチャポイントアイコンとパワーアップアイコンは、ビュー範囲の外側にあってかつ検出可能範囲のときには、ビューの境界に貼り付くスティッキーアイコンとして表示されます。検出可能範囲に入出したときにアイコンをフェードさせる処理は Direct Access API メソッドで実現しています。また、ランタイム中の C++を使った Stage と MovieClip の添付・削除にも、Direct Access API メソッドを使用しています。

この「Minimap」シンボルは複数のレイヤーで構成されています。レイヤー構造は、深さ方向に整列しているエレメントに対してオーサータイムコントロールを行うときに有用です。最上位レイヤーはそのほかのレイヤーの上位に表示され、その他のレイヤーについても同様です。

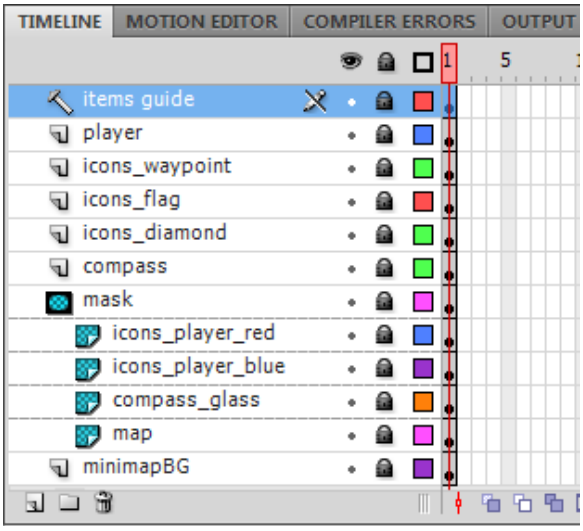

図 4: 「minimap」シンボルのレイヤー構造

- **items guide**: デザイナー／アーティストに便利なガイドレイヤーです。このレイヤー上のコンテンツは SWF ファイルにはパブリッシュされません。
- **player**: 黄色のプレーヤーアイコンを含むレイヤーです
- **icons_waypoint**: ウェイポイントアイコンの作成とキャッシュを行う空のキャンバスです。
- **icons_flag**: フラグアイコンの作成とキャッシュを行う空のキャンバスです。
- **icons_diamond**: パワーアップの菱形アイコンの作成とキャッシュを行う空のキャンバスです。
- **compass**: コンパスを含むレイヤーです（北にマーキングあり）。
- **mask**: 非マスク領域内のコンテンツのみを表示するマスクレイヤーです。
    - **icons_player_red**: 赤色の敵アイコンの作成とキャッシュを行う空のキャンバスです。
    - **icons_player_blue**: 青色の味方アイコンの作成とキャッシュを行う空のキャンバスです。
    - **compass_glass**: コンパスの左上にガラスのような輝きを与える装飾です。
    - **map**: 地形図を含むレイヤーです。このデモでは大きな地形ビットマップを使用していますが、製品では C++から更新するテクスチャがおそらくは使用されるでしょう。この場合は、マスク済み地形図を使用する場合とは違って、更新ロジックでマップのオフセット／回転を変更する必要はありません。

- **minimapBG**: 地形ビットマップのエッジと適切にブレンドされる青の背景で構成されています。背景は地形ビットマップがビューから外れたときの切り替えをスムーズにするためです。C++を使って地形マップをテクスチャとしてレンダリングして更新する場合は、背景は必要ありません。

「Minimap」シンボルは com.scaleform.Minimap クラスを使います（*Bin\Data\AS2 or AS3\Kits\HUD \com\scaleform* に格納）。この最小クラスは、マップイメージへのパスを定義するとともに、HUDKitDemo アプリケーションへの Minimap 登録とマップイメージのロードを行う minimap のコンストラクタを定義します。

4 HUD ビュー

C++ HUD View はコンテンツ HUDKit.swf を管理します。そのインタフェースは次のタイプを宣言します。

- FxHUDView – HUD（とくに、HUDKit.swf）のコアマネージャ。シミュレーションのステートに基いて、ビューとすべてのチャイルド MovieClip を更新します。
- BillboardCache – 味方／敵ビルボードのマネージャおよびキャッシュシステム（中央のオンスクリーンインジケータを minimap アイコンと混同しないようにしてください）。
- FxHUDLog – HUD 左に表示されるメッセージログのマネージャ。FxHUDMessages の追跡、表示、および拒否を行います。
- FxHUDMessage – メッセージ定義で、MovieClip リファレンスとメッセージテキストを含みます。

FxHUDView のキャッシュされた MovieClip リファレンスは、MovieClip の短縮形である *MC* サフィックスによって示される点に注意してください。

FxHUDView は初期化中に、ランタイム時に UpdateView()で変更される主な MovieClip へのリファレンスを登録します。これによって更新ごとの MovieClip リファレンスの検索が不必要になり、結果としてビュー更新の時間が短くなります。また初期化では、HUDKit.fla で定義された未使用 UI エレメントを隠します。

FxHUDView::UpdateView()は、シミュレーション環境が提供するデータを使って、ビューの各エレメントを更新します。そのロジックは、UpdateTeamStats()、UpdatePlayerEXP()、Update PlayerStats()、UpdateEvents()、UpdateTutorial()、UpdateBillboards()という各サブメソッドに分割されます。これらメソッドは、特定 UI エレメントのアップデートに使われるポインタを、シミュレーション環境に渡します。

MovieClip リファレンスは UpdateView()コール中にはほとんど検索されない点に留意してください。MovieClip をフレームごとに更新すると処理が遅くなる可能性があるため、不必要な MovieClip 更新は可能なかぎり省略しています。

Scaleform Direct Access API のリリースにより、いかなるタイプおよび形式のデータも、Flash とアプリケーション間で効率的に受け渡しすることができるようになりました。とくに GFx::Value クラスには、Flash コンテンツを効率的に制御する、数々の新しい機能が搭載されています。HUD View の更新ロジックでは、GFx::Value::SetDisplayInfo()、GFx::Value::GetDisplayInfo()、GFx::Value::SetText()の活用によって、MovieClip の表示ステートの直接かつ効率的な操作を実現しています。

4.1 ラウンドとの統計とスコアボード

HUDKit.fla の Scene 1 内の teamStats レイヤーに配置された *teamStats* MovieClip は、現在進行中のラウンドのスコアと統計情報を表します。この MovieClip は、勝敗の textField（*redWinning*、*blueWinning*）、2 つの textField に表示される両チームのスコア（*scoreRed*、*scoreBlue*）とプログレスバー（*teamRed*、*teamBlue*）、および、ラウンド時間クロックの textField（*roundTime*）で構成されています。

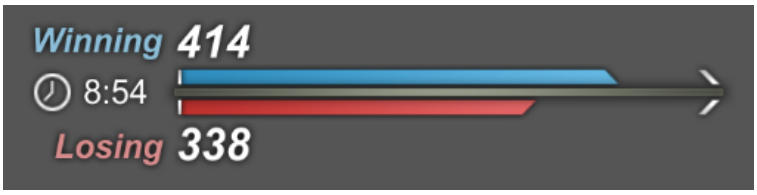

図 5: teamStats シンボル

teamStats は FxHUDView::UpdateTeamStats()を使って排他的に更新されます。以下に *teamStats* MovieClip のエレメントを更新するコードを示します。

```
void FxHUDView::UpdateTeamStats(FxHUDEnvironment *penv)
{
	// Retrieve the scores from the simulation.
	unsigned scoreBlue = unsigned(penv->GetHUDBlueTeamScore());
	unsigned scoreRed = unsigned(penv->GetHUDRedTeamScore());

	String text;
	Value tf;
	Value::DisplayInfo info;

	// We won't update any element who has not changed since the last update
	// to avoid unnecessary updates.  To do so, we compare the previous
	// simulation State to the latest simulation data.  If the two are
	// different, update the UI element in the HUD View.
	if (State.ScoreRed != scoreRed)
	{
		Format(text, "{0}", scoreRed);
		ScoreRedMC.SetText(text); // Update the red team score text.

		info.SetScale((scoreRed / (Double)MaxScore) * 100, 100);
		TeamRedMC.SetDisplayInfo(info); // Update and scale the red team
                                        score bar movieClip.
	}

	if (State.ScoreBlue != scoreBlue)
	{
		Format(text, "{0}", scoreBlue);
		ScoreBlueMC.SetText(text); // Update the blue team score text.

		info.SetScale((scoreBlue / (Double)MaxScore) * 100, 100);
		TeamBlueMC.SetDisplayInfo(info); // Update and scale the blue team
```

```
                                                score bar movieClip.
        }

        if (State.ScoreRed != scoreRed || State.ScoreBlue != scoreBlue)
        {
                // Update the "Winning" and "Losing" text fields for both teams
                // if the score has changed.
                if (scoreBlue > scoreRed)
                {
                        RedWinningMC.SetText(LosingText);
                        BlueWinningMC.SetText(WinningText);
                }
                else
                {
                        RedWinningMC.SetText(WinningText);
                        BlueWinningMC.SetText(LosingText);
                }
        }

        // Update the roundtime clock
        float secondsLeft = penv->GetSecondsLeftInRound();
        unsigned sec = ((unsigned)secondsLeft % 60);
        if (sec != State.Sec)
        {
                unsigned min = ((unsigned)secondsLeft / 60);
                String timeLeft;
                Format(timeLeft, "{0}:{1:0.2}", min, sec);
                RoundTimeMC.SetText(timeLeft);
                State.Sec = sec;
        }
}
```

スコアを Scaleform::String 内に格納するとともに、GFx::Value::SetText()を使って *scoreRed* と *scoreBlue* の textField に渡します。teamRed と teamBlue のスコアバーの X 軸は GFx::Value::SetDisplayInfo()を使って更新します。両方のスコアバーはそれぞれの幅の 0%点から始まり、チームのスコアがラウンドの勝利に必要なスコアに近づくにつれて、100%に向けて伸張します。

blueWinning/redWinning の textField を更新する前に、不必要な更新を避けるために 2 つのチームスコアの比較を行っています。GFx::Value::SetText()をコールし、HUDView の初期化中に定義された「Winning」および「Losing」の固定ストリングを使って、テキストを設定します。

ラウンド時間の textField を更新する前に、最後の UpdateView()コール時点での残り時間と最新の時間との比較を行っています。前回の更新から 1 秒以上が経過している場合、GFx::Value::SetText()を使って、*roundTime* textField をフォーマット済み時間 String で更新します。

4.2 プレーヤー統計とアモ

HUDKit.fla の Scene 1 内の playerStats レイヤー内に配置された playerStats シンボルは、ヘルス textField とヘルスバー形状（healthN、health）、武器アイコンシンボル（weapon）、HUD 用アモ（銃弾数）インジケータ、および低アモインジケータ（reload）で構成されます。

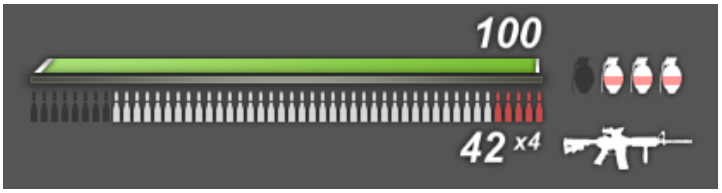

図 6: playerStats シンボル

アモインジケータは、*clipN*、*ammoN*、*ammo*、*ammoBG*、*grenade*、*grenadeBG* という 6 種類の MovieClip に分解されます。textField の *clipN* と *ammoN* は、武器とマガジンクリップの形式で、アモの残量を表示します。

ammo/healthVehicle レイヤーは *ammo* と *ammoBG* の 2 つのシンボルで構成され、それぞれは適切なサブレイヤーに分けられます。両方のシンボルはヘルスバー下のアモインジケータの形状を表します。各シンボルそれぞれは 51 個の Keyframe に分割され、それぞれひとつ前の Keyframe よりも 1 個だけ少ない量の銃弾を示します。GFx::Value::GotoAndStop()はそれら MovieClip の各 Keyframe をアクセスするために使用され、シミュレーションの状態を反映して、表示される図形数を変更します。

- *ammo*は装備済みの武器にすでに装填されている銃弾数を示します
- *ammoBG*は装備済みの武器のクリップに装填できる最大銃弾数を示します。

grenade/armor レイヤーは、アモレイヤーと同様に、それぞれが独自のレイヤーに分離される *grenade* と *grenadeBG* の 2 個のシンボルによって構成されます。両方のシンボルはグレネードをインジケータ図形で表します。アモインジケータと同様に、各シンボルは 5 個の Keyframe に分割されます。Frame 1 には 4 個グレネード図形が収められ、Frame 5 には図形はありません。GFx::Value::GotoAndStop()はそれら MovieClip の各 Keyframe をアクセスし、シミュレーションの状態を反映して、表示される図形の個数を変更します。

- *grenade*は残りのグレネード数を示します。
- *grenadeBG*はプレーヤーが携行できるグレネード数を示します。

以下に *playerStats* シンボルを更新するロジックを示します。

```
void FxHUDView::UpdatePlayerStats(FxHUDEnvironment *penv)
{
      FxHUDPlayer* pplayer = penv->GetHUDPlayer();

      // Load latest player info.
```

```
unsigned ammoInClip = pplayer->GetNumAmmoInClip();
unsigned ammoClips  = pplayer->GetNumClips();
unsigned clipSize = pplayer->GetMaxAmmoInClip();
unsigned grenades = pplayer->GetNumGrenades();

unsigned ammoFrames = 51;
unsigned grenadeFrames = 5;

String text;

if(pplayer->GetHealth() != State.Health)
{
      unsigned health = unsigned(pplayer->GetHealth() * 100);
      Format(text, "{0}", health);
      HealthNMC.SetText(text); // Update the health text field.

      Value::DisplayInfo info;
      info.SetScale(Double(health), 100);
      HealthMC.SetDisplayInfo(info); // Update and scale the health bar.
}

if(ammoInClip != State.AmmoInClip)
{
      Format(text, "{0}", ammoInClip);
      AmmoNMC.SetText(text); // Update the ammo text field.

      // The ammo is divided into two parts: the white bullet icons (AmmoMC)
      // and their respective backgrounds (AmmoBGMC).

      // To update the number of bullets currently in the clip, we use
      // GotoAndStop with AmmoMC and select the frame with the proper
      // number of white bullet icons.  There are 51 frames in AmmoMC
      // (defined above in ammoFrames)and the first frame displays every
      // bullet icon. To display X number of bullets, we subtract X from
      // the total number of frames in AmmoMC (51) and GotoAndStop on the
      // result.
      AmmoMC.GotoAndStop(ammoFrames - ammoInClip);
}

if(ammoClips != State.AmmoClips)
{
      Format(text, "x{0}", ammoClips);
      ClipNMC.SetText(text); // Update the remaining ammo clips text field.
}

if(grenades != State.Grenades)
{
```

```
            // Grenades are setup similar to Ammo.  The first frame for the
            // Grenades movieClip shows 4 white grenade icons.  The second frame
            // shows 3.

            // To display X number of grenades, we subtract X from the total
            // number of frames in GrenadeMC (5) and GotoAndStop on the result.
            // Note: GrenadeMC actually contains 10 frames but 6-10 are unused by
            // this implementation.
            GrenadeMC.GotoAndStop(grenadeFrames - grenades);
      }

      if (pplayer->GetWeaponType() != State.weaponType)
      {
            // Change the weapon icon if the player has changed weapons.
            unsigned weaponFrame = GetWeaponFrame(pplayer->GetWeaponType());
            WeaponMC.GotoAndStop(weaponFrame);

            // Update the ammo background icons based on clipSize.
            if (clipSize != State.AmmoClipSize)
                  AmmoBGMC.GotoAndStop(ammoFrames - clipSize);
      }

      // If the ammo in clip is less than 6 bullets, play the "Reload" indicator
      // movieClip.
      if (ammoInClip <= 5)
      {
            // ReloadMC will play until we GotoAndStop on Frame 1, so no need to
            //  start the animation more than once.
            if (!bReloadMCVisible)
            {
                  bReloadMCVisible = true;
                  ReloadMC.GotoAndPlay("on"); // Will cycle back to "on" frame
                                              when it reaches the end and play.
                                              again
            }
      }

      // If the reload indicator is displayed but there are more than six bullets
      // hide it and stop the animation by playing frame 1 of the movieClip.
      else if (bReloadMCVisible)
      {
            ReloadMC.GotoAndStop("1"); // Frame 1 contains stop();
            bReloadMCVisible = false;
      }
}
```

ammo において、Frame 1 は 50 発の銃弾インジケータで、Frame 51 は 0 発の銃弾インジケータです。プレーヤーが武器を発砲したとき、適切な Keyframe をアクセスするために ammoMC.GotoAndStop(ammoFrames - ammoInClip)がコールされて、HUD から装填済み銃弾インジケータを 1 個除去します。

ammoBG の場合、ammoBGMC.GotoAndStop()を使って、X 個の銃弾インジケータをバックグラウンドに表示している Keyframe をアクセスします。ここで X は装備済み武器のクリップサイズです。たとえばマシンガンのクリップサイズは 50 です。ゆえに ammoBG.GotoAndStop(1)は、50 発の銃弾インジケータすべてを Frame 1 のバックグランドに表示します。この MovieClip は、装備済み武器が最後の FxHUDView::UpdateView()コールから変更されたときのみ更新されます。

4.3 フラグキャプチャインジケータ

HUDKit.fla の Scene 1 の flag レイヤー内に配置された *flagMC* シンボルは、フラグキャプチャインジケータのコンテナです。*flagMC* は、フラグキャプチャインジケータビットマップと、フラグの CaptureState に応じて左または右に移動するアローインジケータ（*arrow*）の 2 つのパーツに分割される *flag* シンボルで構成されます。インジケータは、プレーヤがキャプチャオブジェクトに近づくとフェードインし、プレーヤがレンジ外に出るとフェードアウトします。

図 7: flagMC シンボル

flagMC のフェードインおよびフェードアウトアニメーションは、Classic Tween を使った Flash タイムライン上で実行されます。*flagMC* のタイムラインは、5 個の Keyframe によって表現される Hidden、Visible、Fade-In、FadeOut という 4 種類の図形ステートで構成されます。これらステートは GotoAndStop()と Keyframe の labels/numbers（1、5、“On”、“Off”、その他）を使ってアクセスします。

以下に示す FxHUDView::UpdateEvents()は、*flagMC* のステートをコントロールし、シミュレーションのデータを使って *arrow* 位置を更新します。

```
// Flag capture indicators and reticule behavior.
void FxHUDView::UpdateEvents(FxHUDEnvironment *penv)
{
      Value::DisplayInfo info;

      if (penv->IsHUDPlayerCapturingObjective())
      {
            if(!bFlagMCVisible)
            {
                  SetVisible(&FlagMC, true); // Set the Flag MovieClip's
                                             visibility to true.
                  FlagMC.GotoAndPlay("on"); // Play the fade-in animation once.
                  bFlagMCVisible = true;
            }

            // Shift the x-coordinate of the Flag Arrow to show the capture state
            // of the flag the player is currently capturing.
            info.SetX(penv->GetHUDCaptureObjectiveState() * 177);
            FlagArrowMC.SetDisplayInfo(info);
            info.Clear();
```

```
        }
        else if (bFlagMCVisible & !penv->IsHUDPlayerCapturingObjective())
        {
                // If the flag indicator is still visible and the player is
                // no longer capturing an objective, play the fade-out animation
                FlagMC.GotoAndPlay("off");
                bFlagMCVisible = false;
        }
}
```

flagMC.GotoAndPlay("on")がコールされたとき、flagMC はフェードインアニメーションを再生したあと、Visible ステートで停止します。flagMC.GotoAndPlay("off")がコールされると、flagMC はフェードアウトアニメーションの Frame 11～20 を再生します。アニメーションが Frame 20 に達すると、Frame 1 に自動的に戻って再生を繰り返します。このような動作は Flash のビヘイビアとしてはデフォルトですので、ActionScript また C++を追加する必要はありません。Frame 1 には stop();コールが含まれるため、MovieClip は Frame 1 を再生したあとで停止します。Frame 1 では Alpha を 0 に設定してシンボルを隠しているため、このような動作は理想的と言えるでしょう。

arrow は GFx::Value::SetDisplayInfo()によって水平方向にシフトします。この方法は、エンティティの CaptureState に基づく更新済み X 座標を使って、新しい GFx::Value::DisplayInfo に渡されます。

4.4 武器のレティクル

HUDKit.fla の Scene 1 の crosshair レイヤー内に配置されている *reticule* シンボルは発砲レティクルのコンテナです。*reticule* は、それぞれが専用のレイヤーを持つ、*left*、*right*、*bottom*、*top* の 4 つのシンボルで構成されます。それぞれのシンボル内にレティクルの各断片を持たせることで、個別の操作が可能になっています。ユーザープレーヤーが武器を発砲すると、レティクルはシミュレーションによってトリガーされる PlayerFire イベントに応答します。武器の発砲中はレティクルを動的に拡大します。プレーヤーが発砲を停止するとレティクルは元の大きさに戻ります。

図 8: レティクルシンボル

レティクルの更新コードは FxHUDView::UpdateEvents()メソッドと FxHUDView::OnEvent()メソッドの 2 つに分割されます。ReticuleHeat と ReticuleFiringOffset は、PlayerFire イベントを捕捉した FxHUDEvent にて、それぞれインクリメントおよび設定が行われます。

```
// If the player is firing his/her weapon, update the reticule's heat.
// The reticule elements are updated in FxHUDView::UpdateEvents() based on
// the ReticleHeat and the ReticleFiringOffset.
case FxHUDEvent::EVT_PlayerFire:
{
      if (ReticleHeat < 8)
            ReticleHeat += 2.5f;

      ReticleFiringOffset = 2;
}
```

FxHUDView::UpdateEvents()はこれら 2 個の変数を使って、*top*、*bottom*、*left*、*right* の各 MovieClip をシフトします。各 MovieClip では、SetDisplayInfo()を使って MovieClip の X または Y 軸をシフトしています。式の最初の値である 6 または-6 は、座標（0,0）からの開始 X または開始 Y オフセットです。SetDisplayInfo() コールでは GFx::Value::DisplayInfo のシングルインスタンスが再利用されます。ただし、DisplayInfo が意図せず再利用されないように、各 GFx::Value::SetDisplayInfo()コールのあとに DisplayInfo.Clear()がコールされます。この方法によって DisplayInfo インスタンスに過去に定義されたすべての表示プロパティがクリアされます。

```
// Shift the XY-coordinates of the reticule elements based on the
// firing duration and rate of fire.
if (ReticleHeat > 0 || ReticleFiringOffset > 0){
      if (ReticleHeat > 1.0f)
```

```
            ReticleHeat -= .02f;

        ReticleFiringOffset -= .02f;
        if (ReticleFiringOffset < 0)
            ReticleFiringOffset = 0;

        info.SetY((-6) - (ReticleFiringOffset + ReticleHeat));
        ReticuleMC_Top.SetDisplayInfo(info);
        info.Clear();

        info.SetX((6) + (ReticleFiringOffset + ReticleHeat));
        ReticuleMC_Right.SetDisplayInfo(info);
        info.Clear();

        info.SetX((-6) - (ReticleFiringOffset + ReticleHeat));
        ReticuleMC_Left.SetDisplayInfo(info);
        info.Clear();

        info.SetY((6) + (ReticleFiringOffset + ReticleHeat));
        ReticuleMC_Bottom.SetDisplayInfo(info);
        info.Clear();
}
```

このデモでは、処理を簡略化するために、単純かつ動的なレティクル式を選択していますが、よりダイナミックかつクリエイティブなレティクルビヘイビアを得るには、このコードをカスタマイズしてください。

4.5 方向性ヒットインジケータ

HUDKit.fla の Scene 1 内の dirHit レイヤー内に配置された *dirHit* シンボルは、ディレクションヒットインジケータのコンテナです。*dirHit* は、適切な名前のレイヤーに分割される、*tl*、*l*、*bl*、*b*、*br*、*r*、*tr*、*t* という 8 個の MovieClip で構成されています。

それぞれの MovieClip には、以下のインジケータとして使用される、アルファブレンドの赤色セミサークルが含まれます。

a) ユーザープレーヤーがダメージを受けた
b) そのダメージの原因となった相対方向

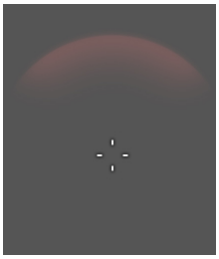

図 9: dirHit シンボル

プレーヤが打たれると、ヒットインジケータが現れ、およそ 1 秒後にフェードアウトします。フェードアウトアニメーションは Flash タイムライン上の Classic Tween アルファブレンドです。ヒットインジケータの MovieClip のいずれかを GFx::Value::GotoAndPlay(“on”)でコールすることで、アニメーションが現れ 1 秒後にフェードアウトします。適切な MovieClip は、ダメージの原因となった相対方向で決まります。

```
void FxHUDView::OnEvent( FxHUDEvent* pevent )
{
      switch ( pevent->GetType() )
      {
            // If the Damage Event (Player was hit) is fired, show the
            // appropriate directional hit indicator.
            // We can use GotoAndPlay("on") to play the fade-in animation.  It
            // will fade out on its own  after ~1 second (this animation is setup
            // on the Flash timeline)
            case FxHUDEvent::EVT_Damage:
            {
                  FxHUDDamageEvent* peventDamage = (FxHUDDamageEvent*)pevent;
                  float dir = peventDamage->GetDirection();
                  if (dir > 0)
                  {
                        if (dir > 90)
                        {
```

```
                        if (dir > 135)
                              DirMC_B.GotoAndPlay("on");
                        else
                              DirMC_BR.GotoAndPlay("on");
                  }
                  else
                  {
                        if (dir > 45)
                              DirMC_TR.GotoAndPlay("on");
                        else
                              DirMC_T.GotoAndPlay("on");
                  }
            }
            else
            {
                  if (dir < -90)
                  {
                        if (dir < -135)
                              DirMC_B.GotoAndPlay("on");
                        else
                              DirMC_BL.GotoAndPlay("on");
                  }
                  else
                  {
                        if (dir < -45)
                              DirMC_L.GotoAndPlay("on");
                        else
                              DirMC_TL.GotoAndPlay("on");
                  }
            }
            break;
      }
}
```

4.6 ランクと経験値の表示

HUDKit.fla の Scene 1 の expBar レイヤー内に配置された *expBar* シンボルは、プレーヤーの現在のランキングをトラッキングする、画面下の exp バーを表します。*expBar* は、プレーヤーがキルとキャプチャで獲得した経験値に伴って増加する黄色バーの *exp* MovieClip で構成されます。

HUDKit.fla の Scene 1 の rank レイヤー内に配置された *rank* シンボルは、プレーヤーの現在のランキングを表します。*rank* はそれぞれひとつのランクに対応した 18 個の Keyframe で構成されます。各 Keyframe は、ランキングに応じたアイコンとタイトルを表示します。

図 10: expBar と rank シンボル

```
void FxHUDView::UpdatePlayerEXP(FxHUDEnvironment *penv)
{
      FxHUDPlayer* pplayer = penv->GetHUDPlayer();
      if (pplayer->GetLevelXP() != State.Exp)
      {
            Value::DisplayInfo info;
            // Update the rank icon.  Each rank icon is on a different frame of
            // the rank movieClip.
            RankMC.GotoAndStop(UInt(pplayer->GetRank() + 1));
            info.SetScale(pplayer->GetLevelXP() * 100, 100);  // Scale the EXP
                                                                 bar movieClip.
            ExpBarMC.SetDisplayInfo(info);
      }
}
```

チームスコアバーと同様に、プレーヤーの経験値が変更されると、*exp* はプレーヤーの現在の経験値に応じて拡大または縮小が行われます。*rank* を更新するには、GFx::Value::GotoAndStop()を Keyframe X に使用します。ここで X はプレーヤーの現在のランク＋1 です（ゆえに Frame 0 はありません）。

4.7 テキスト通知

HUDKit.fla の Scene 1 の centerText レイヤー内に配置された *centerTextMC* シンボルは、画面中央でユーザーに情報を示すために使われるアニメーション付き textField です。このデモで *centerText* は、各ラウンドの開始時点で簡単なチュートリアルを表示するためと、ユーザープレーヤーがキルされたときに使われます。テキストは表示後にフェードアウトします。

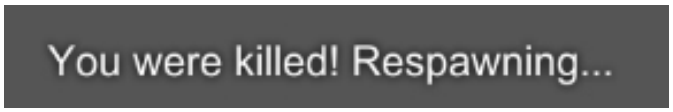

図 11: centerTextMC シンボル

centerTextMC は、実際の textField である *textField* で構成される、*centerText* MovieClip から構成されています。MovieClip はアニメーションができるように整列が行われます。*centerTextMC* のタイムラインは、フェードアウトアニメーションを行う前にテキストを 5 秒間表示する「on5」と、フェードアウトアニメーションを行う前にテキストを 3 秒間表示する「on」という、2 個のラベル付き Keyframe で構成されます。タイムラインが終了すると、*centerTextMC* のアルファが 0 に設定されている Frame 1 から MovieClip のプレイバックが再開され、「on」または「on5」が再コールされるまで非表示となります。

```
TextFieldMC.SetText(RespawnText); // Set the text field of the Center Text.
CenterTextMC.GotoAndPlay("on"); // Play the fade-in animation, it will fade on
                                   its own after ~3 sec.
```

ここで、*centerTextMC* の *textField* は適切に設定され、MovieClip は GotoAndPlay(“on”)を使って表示されます。この MovieClip は、タイムラインの Classic Tween によって、およそ 3 秒後に自らをフェードアウトします。

4.8 ポップアップと通知

HUDKit.fla の Scene 1 の popup レイヤー内に配置された *popup* シンボルは、アニメーション付きのイベントインジケータです。このデモで *popup* は、ユーザープレーヤーが 3 回以上連続してキルされたときに再生されます（キル・ストリーク）。表示される場合はフェードイン後にアニメーションが再生され、C++コールにもとづいて、3～5 秒後にフェードアウトします。

図 12: popup シンボル

Popup はポップアップのコアコンテナです。そのサブエレメントのすべてのアニメーションは、Classic Tweens を使った *popup* のタイムライン上で行われます。*popup* は、*actions*、*popupNumber*、*popupText*、*popupBG* の 4 つのレイヤーで構成されています。*popupBG* は数字の背景で「爆発」しているアニメーション付きの黒い図形です。*popupText* と *popupNumber* は、ポップアップメッセージのテキストと数字を表示する *textField* シンボルで構成されています。タイムライン上で textField のアニメーションを行うには、このようなシンボル階層が必要です。

シンボルは「on」の Keyframe を持っていて、GotoAndPlay()を使ってコールするとアニメーションが始まります。初期アニメーションが完了すると、ポップアップはフェードアウトして、アルファが 0 に設定されていて非表示となる Frame 1 に戻り停止します。

```
// If a KillStreak event is fired, display the kill streak pop-up.
case FxHUDEvent::EVT_KillStreak:
{
      FxHUDKillStreakEvent* peventKS = (FxHUDKillStreakEvent*)pevent;

      // Format the number of kills
      String text;
      Format(text, "{0}", peventKS->GetSrcKillStreak());
      PTextFieldMC.SetText(text); // Set the text field of the PopUp.

      // Play the fade-in animation, it will fade on its own after ~4 sec.
      PopupMC.GotoAndPlay("on");
}
break;
```

ここでは KillStreak イベントとして FxHUDEvent をキャストしています。続いて、プレーヤーから連続キル回数を読み出し、適切にフォーマットします。あらかじめキャッシュしておいた *popupText* の *textField* シンボルである PTextFieldMC のテキストを C++で設定し、GotoAndPlay(“on”)を使って *popup* のディスプレイアニメーションをトリガーしています。

4.9 メッセージとイベントログ

HUDKit.fla の Scene 1 の log レイヤー内に配置された *log* シンボルは、HUD の左下に表示されるメッセージとイベントログのコンテナです。このデモでログは、キル、パワーアップ、フラグキャプチャ、レベルアップといった最近のイベントに関連するテキストメッセージを表示します。新しいメッセージはログの下に追加され、古いメッセージは上に押しやられます。ログメッセージはおよそ 3 秒後にフェードアウトします。

図 13: log と logMessage シンボル

ログメッセージは *logMessage* シンボルのインスタンスです。ログに追加されたメッセージは、フェードアウトアニメーションが始まる前におよそ 3 秒間表示されます。このアニメーションは *logMessage* シンボルのタイムライン上で定義されています。メッセージテキストは C++から送られ、htmlText を使って textField に表示されます。

FxHUDMessage::Movieclip* FxHUDLog::GetUnusedMessageMovieclip()で始まる以下のコードは、新しい LogMessage MovieClip を作成し、その MovieClip を、すべてのログメッセージのキャンバスである Log MovieClip に添付します。

```
FxHUDMessage::Movieclip* FxHUDLog::GetUnusedMessageMovieclip()
{
    if (MessageMCs.IsEmpty())
    {
      // Request a new line in the SWF
      Value logline;
      LogMC.AttachMovie(&logline, "LogMessage", "logMessage"+NumMessages);
      FxHUDMessage::Movieclip* pmc = new FxHUDMessage::Movieclip(logline);
      MessageMCs.PushBack(pmc);  // Add new Message MC to list of unused
                                      message movieClips.
    }
    FxHUDMessage::Movieclip* pmc = MessageMCs.GetFirst();  // Use an unused
                                                              Message movieClip.
    pmc->SetVisible(true);
    MessageMCs.Remove(pmc);
    return pmc;
```

```
}
```

メッセージのコンテンツはイベントタイプに基づいて C++から設定します。以下は、Level Up イベントを処理するとともに、ログ中の「You are now a [Rank Icon] [Rank Name]」を表示メッセージとして htmlText に設定する C++コードです。

```
void    FxHUDLog::Process(FxHUDEvent *pevent)
{
      String msg;
      switch (pevent->GetType())
      {
            case FxHUDEvent::EVT_LevelUp:
            {
                  FxHUDLevelUpEvent* e = (FxHUDLevelUpEvent*)pevent;
                  Format(msg, "You are now a <img src='rank{1}'/> {0}!",
                  e->GetNewRankName(), e->GetNewRank());
            }
            break;
      }

FxHUDMessage::Movieclip* pmc = GetUnusedMessageMovieclip();
FxHUDMessage* m = new FxHUDMessage(msg, pmc); // 3 seconds
Log.PushBack(m);
NumMessages++;
```

以下のコードはログの管理に使用する HUD Update ロジックです。このコードはログメッセージリファレンスのレイアウトとアニメーションを更新します。

```
void    FxHUDLog::Update()
{
      SF_ASSERT(LogMC.IsDisplayObject());

      Double rowHeight = 30.f;

      // Tick the message lifetimes
      Double yoff = 0;
      FxHUDMessage* data = Log.GetFirst();
      while (!Log.IsNull(data))
      {
      FxHUDMessage* next = Log.GetNext(data);
            if (data->IsAlive())
            {
            // Layout mgmt and animation
            Value::DisplayInfo info;
```

```
            info.SetY(yoff);
            data->GetMovieclip()->GetRef().SetDisplayInfo(info);
            }
            data = next;
            yoff += rowHeight;
      }

      // Layout management and animation
      Value::DisplayInfo info;
      info.SetY(LogOriginalY - (NumMessages * rowHeight));
      LogMC.SetDisplayInfo(info);

      // Remove dead messages
      data = Log.GetFirst();
      while (!Log.IsNull(data))
      {
            FxHUDMessage* next = Log.GetNext(data);
            if (!data->IsAlive())
            {
            Log.Remove(data);
            NumMessages--;

            FxHUDMessage::Movieclip* pmc = data->GetMovieclip();
            pmc->SetVisible(false);
            MessageMCs.PushBack(pmc);

            delete data;
            }
            data = next;
      }
}
```

Log の更新ロジックでは、その Alpha が 0 に設定されているかどうかをチェックして、いずれかの FxHUDMessage が現在表示されているかを確認しています。表示されていない場合は Log から削除し、未使用の FxHUDMessage MovieClip の MessageMC リストに戻しています。新しい FxHUDMessage が追加された場合は、GFx::Value::DisplayInfo.SetY()と GFx::Value::SetDisplayInfo()を使って、すべての MessageMC を適切にシフトアップしています。

4.10 ビルボード

HUDKit.fla の Scene 1 の billboard レイヤー内に配置された *billboard_container* は、プレイヤービルボードのコンテナです。各ビルボードはプレーヤの位置のあとに矢印とプレーヤー名を表示します。ビルボードは、ゲーム中で、プレーヤーのビュー内に含まれる 3D オブジェクトの詳細表示によく使われます。このデモでビルボードは、ユーザープレーヤーから特定の距離に現れます。味方プレーヤービルボードにはプレーヤー名が必ず表示されますが、敵ビルボードにはターゲットがビューの中心に近づくまでは矢印しか表示されません。

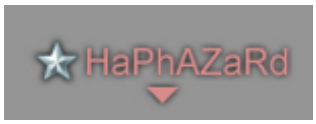

図 14: billboard_enemy シンボル

このデモでのビルボード実装では、GFx::Value::SetDisplayInfo()を使って移動および拡大縮小を行いプレーヤー位置に追随するようにした、2D ビルボードの MovieClip を UI 内に作成しています。

各ビルボードの MovieClip（*billboard_friendly*、*billboard_enemy*）は、text レイヤーに配置されたひとつの *billboard_label_*（*friendly/enemy*）と、arrow レイヤーに配置されたひとつの矢印イメージで構成されています。*billboard_label_** は対象プレーヤー名の表示に使われる textField で構成されます。*billboard_friendly* の「Show」Keyframe は、*billboard_friendly* タイムライン上の Classic Tween アルファブレンドを使って。フェードインアニメーションを開始します。「Show」が再生されると、非表示または削除が行われるまで、ビルボードは表示された状態を維持します。

ビルボード MovieClip をキャンバスに添付する C++ロジックは FxHUDView::BillboardCache::GetUnusedBillboardMovieclip に収められています。

```
if (UnusedBillboards.IsEmpty())
{
      Value::DisplayInfo info(false);

      Value billboard, temp;
      String instanceName;
      Format(instanceName, "{0}{1}", BillboardPrefix, BillboardCount++);
                  CanvasMC.AttachMovie(&billboard, SymbolLinkage, instanceName);

      BillboardMC* pbb = new BillboardMC();
      pbb->Billboard = billboard;
      billboard.GetMember("label", &temp);
      temp.GetMember("textField", &temp);
      pbb->Textfield = temp;

      pbb->Billboard.SetDisplayInfo(info);
```

```
        UnusedBillboards.PushBack(pbb);

}
```

各 BillboardCache には、CanvasMC という名前のビルボードの添付先である MovieClip へのリファレンスが収められています。このリファレンスはインスタンス化されたときに BillboardCache へ渡されます。上のロジックでは、シンボル billboard_enemy/billboard_friendly のビルボード MovieClip を、CanvasMC.AttachMovie()コールの時点で定義済みキャンバス MovieClip に添付し、再利用のためにリファレンスを保持します。

Billboard 更新ロジックは、次の UpdateBillboards()、BeginProcessing()、EndProcessing()、GetUnusedBillboardMovieclip()の各メソッドで定義されます。BeginProcessing()と EndProcessing()は、味方ビルボードセットまたは敵ビルボードセットの更新の開始時または終了時にコールされます。これらメソッドは使用および未使用のビルボードリストを管理します。

以下のコードは UpdateBillboards()から呼ばれる味方ビルボードの更新ロジックです。

```
// Process friendlies
// Friendly billboards will always show names

BillboardMC* pbb = NULL;

FriendlyBillboards.BeginProcessing();
for (unsigned i=0; i < friendlies.GetSize(); i++)
{
      info.Clear();
      if (FriendlyBillboards.GetUnusedBillboardMovieclip(friendlies[i].pPlayer,
                                                         &pbb))
      {
      info.SetVisible(true);
            pbb->Billboard.GotoAndPlay("show");
            // Load player info
            String title;
            Format(title,
                        "<img src='rank{0}' width='20' height='20' align='baseline'
                        vspace='-7'/> {1}",
                        friendlies[i].pPlayer->GetRank()+1,
                        friendlies[i].pPlayer->GetName());
            pbb->Textfield.SetTextHTML(title);
      }
      info.SetX(friendlies[i].X);
      info.SetY(friendlies[i].Y);
      info.SetScale(friendlies[i].Scale, friendlies[i].Scale);
      pbb->Billboard.SetDisplayInfo(info);
}
```

```
FriendlyBillboards.EndProcessing();
```

UpdateBillboards()は、ユーザープレーヤービュークエリから取得したエンティティリストにもとづいて、ビルボードの作成と更新を行います。UpdateBillboards() は、ビルボードの表示には GotoAndPlay(“show”)を使用し、MovieClip の X、Y、および倍率の設定には GFx::Value::SetDisplayInfo()を使用し、テキスト表示には GFx::Value::SetText()を使用しています。

textField は htmlText を使ってビルボードへの入力と表示を行います。このテキストフィールドは、プレーヤーランクのイメージとプレーヤー名で構成されます。ランクアイコンは *src='rank{0}'* HTML で定義されます。上の例では味方プレーヤーのランク＋1 に対応します（friendlies[i].pPlayer->GetRank()+1）。ほかの *img* オプションは、アイコンイメージの高さ、幅、整列、および vspace を定義します。プレーヤー名は{1}で定義されます。上の例では味方プレーヤーの名前が相当します（friendlies[i].pPlayer->GetName()）。

5 Minimap ビュー

Minimap の C++側は次のファイルで構成されます（*Apps\Kits\HUD*に格納）：

- **FxMinimap.h**: minimap デモコア内にゲーム環境とアプリケーションが実装できるインタフェースを宣言します。
- **FxMinimapView.h/.cpp**: minimap ビューで使われるタイプを与えます。

5.1.1Minimap ビュー

minimap ビューインタフェースは以下のタイプを宣言します。

- **FxMinimapIconType**: アイコンタイプの定義で、タイプ ID と特殊化に必要な追加サブタイプ値を含みます。
- **FxMinimapIcon**: ビュー内のエレメントにリンクされるアイコンリソースです。
- **FxMinimapIconCache**: 特定タイプのアイコンセットを格納するアイコンキャッシュ。アイコンビューステートは必要に応じてキャッシュによって管理されます。キャッシュは検出可能範囲の入出でのアイコンのフェードイン／フェードアウトをサポートします。キャッシュは新アイコンの作成にファクトリを使用します。ファクトリの基本実装は MinimapIconFactoryBase と呼ばれるテンプレート化クラスです。各ファクトリは、適切なキャンバス MovieClip にアイコンの MovieClip の添付を必要に応じて行うとともに、minimap が破壊されたとき、それら MovieClip を削除します。以下に示すさまざまなアイコンタイプをサポートするために、複数のファクトリタイプが定義されます。各アイコンタイプは、変換、回転、textField など、アイコンタイプに固有の更新ロジックを実装する Update()メソッドを有します。
    - *PlayerIcon*: 味方アイコンと敵アイコンの両方のリソースを表現します。両方アイコンタイプは同じロジックを共有し、アイコンの生成メソッドのみが異なります。
    - *FlagIcon*
    - *ObjectiveIcon*
    - *WaypointIcon*
- **FxMinimapView**: minimap ビューのメインコントローラ。その UpdateView()メソッドはアプリケーションからコールされ、ビューをリフレッシュします。

以下のコードは PlayerIcon::Update()メソッドです（FxMinimapView.cpp、116 行目）。ビュー内のプレーヤーアイコンを Direct Access API を使って更新するロジックを示しています。*MovieClip* メンバーはステージ上のアイコン MovieClip へのリファレンスで構成されます。SetDisplayInfo メソッドは、性能が遅い SetMember()メソッドを介する代わりに、MovieClip の表示プロパティ（ディスプレイマトリックス、可視フラグ、アルファ値）を直接変更します。なお GFx::Value::SetMember のほうが GFx::Movie::SetVariable よりも高速です。

```
virtual void    Update(FxMinimapView* pview, FxMinimapEntity* pentity,
```

```
                                                        float distSquared)
{
      SF_UNUSED(distSquared);
      bool hasIcon = (pentity->GetIconRef() != NULL);
      PointF pt = pentity->GetPhysicalPosition();
      bool showDir = pview->IsShowingPlayerDir();

      // If entity did not have an icon attached before, then wait for the
      // next update to set the state data. picon is most probably not
      // initialized (the movie needs to advance after creation)
      if (hasIcon && (State != (int)showDir))
      {
            // state 0: no arrow, state 1: show arrow
            Value frame(Double(showDir ? 2.0 : 1.0));
            MovieClip.Invoke("gotoAndStop", NULL, &frame, 1);
            State = (SInt)showDir;
      }

      pt = pview->GetIconTransform().Transform(pt);
      Value::DisplayInfo info;
      if (State)
      {
            info.SetRotation(pview->IsCompassLocked() ?
                  (pentity->GetDirection() +  90)  : (pentity->GetDirection() –
                                                      pview->GetPlayerDirection()));
      }
      info.SetPosition(pt.x, pt.y);
      MovieClip.SetDisplayInfo(info);

}
```

6 性能解析

アプリケーションが表示する統計情報は、HUD ビューの更新に費やした時間、minimap の更新に費やした時間、更新した minimap オブジェクト数、Scaleform コンテンツの総表示時間、および、SWF の先読みに費やした時間を表しています。

オブジェクト数とは、味方プレーヤー/敵プレーヤー、パワーアップ、および目標物など、すべてのエンティティタイプの累積です。HUD の更新時間には、ビルボード、ログメッセージ、ラウンドスコアボード、プレーヤーの統計情報など、すべての UI インジケータの更新時間が含まれます。minimap の更新時間には、前述の minimap オブジェクトの更新に費やした時間のほか、マスク地形、コンパス、プレーヤーアイコンなどの更新時間が含まれます。これらビューの更新には、x/y ポジション、ラベル、現在のフレーム、回転、拡大縮小、マトリックス変換など、MovieClip のビジュアルプロパティで支配されるすべてのロジックの実行が含まれます。

6.1 性能統計

以前の Scaleform では、Movie のビューステートの更新は、C++ Scaleform::Movie::SetVariable と GFx::Movie::Invoke メソッドのみに限られていました。そのため、MovieClip パスの解決など多くの要因によって、大きな性能ハンディを抱えていました。新たにサポートされた Direct Access API は、同じ機能を高速なコードパスで実行するため、Scaleform 3.1 以降を使うことで複雑な HUD 更新をより効率良く行えます。

以下のグラフは、minimap の更新を対象に、Direct Access API メソッドを使ったときの大幅な性能向上効果（更新時間の短縮）と、従来の SetVariable/Invoke メソッドの性能とを比較したものです。10 倍から 25 倍の性能向上が得られていることが分かります。

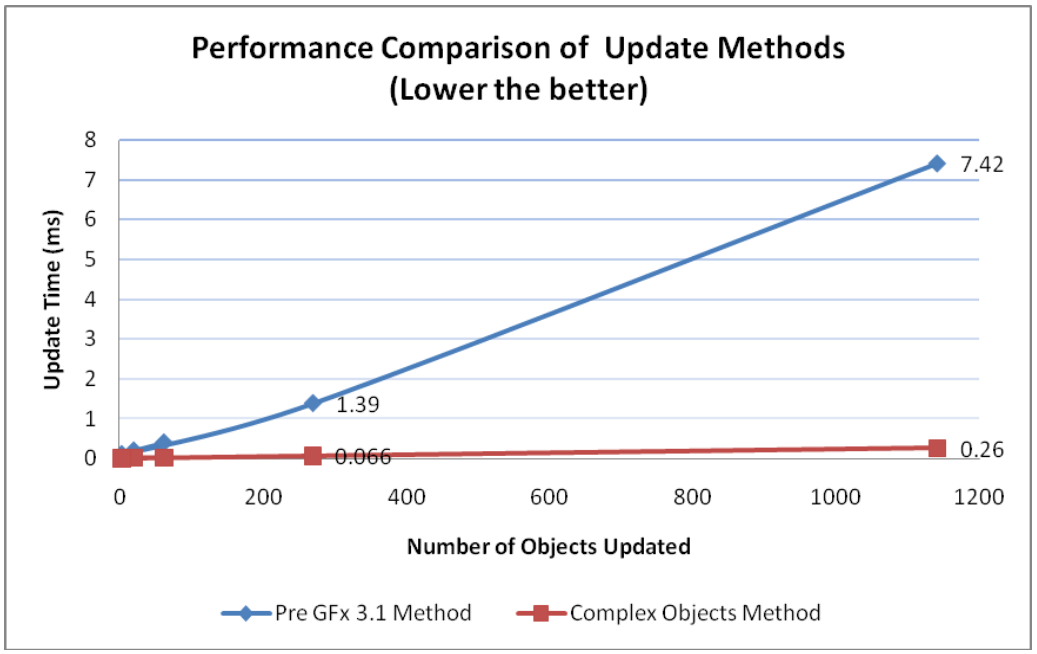

図 15: minimap を対象とした Direct Access API と SetVariable/Invoke との更新性能の比較

上に示すように Direct Access メソッドによる更新時間は、どの条件でも 1ms（HUD UI の平均割り当て処理時間）というスレッショルドを下回っているため、あらゆるワーキング環境に適していることがわかります。

一回 15 分間のシミュレーションラウンドコースの HUD キット全体の平均性能統計をプラットフォームごとに以下に示します。フル機能の Flash HUD を Scaleform で開発した場合の性能見積もりに活用可能です。

プラットフォーム	FPS	更新(ms)	表示(ms)	先読み(ms)	合計(ms)
Windows Vista MacBook Pro 上	1148	0.018	0.512	0.017	0.527
Xbox 360	1136	0.032	0.454	0.010	0.497
PlayStation 3	752	0.044	0.668	0.021	0.733

表 1: 各プラットフォームにおける Scaleform HUD キットの平均性能統計

表 1 に示すように HUD の更新時間は、どの条件でも 1ms（HUD UI の平均割り当て処理時間）というスレッショルドを下回っているため、あらゆるワーキング環境に適していることがわかります。

6.2 メモリの分析

HUD キットのデモでロードされるすべての非圧縮コンテンツのメモリの内訳を以下に示します。コンテンツは GFxExport のデフォルト設定を使ってエクスポートしています。メモリフットプリント全体は、.GFX ファイル、イメージ、コンポーネント、エクスポートフォントで構成されます。

1. **ファイルフォーマット : GFX スタンダードエクスポート、イメージ非圧縮**

全メモリ*:* *1858 KB*

- HUDKit.gfx - 33 KB
- Minimap.gfx - 50 KB
- fonts_en.gfx - 46 KB
- gfxfontlib.gfx - 100 KB

非圧縮イメージ

- HUDKit.gfx
  - コア UI コンポーネント - 603 KB
  - ランキングアイコン - 72 KB
  - 武器アイコン - 51 KB
    - **備考:**このバージョンの HUD キットでは武器アイコンのうち 43 KB は使用していません
- Minimap.gfx
  - コンポーネント - 644 KB
- Map.jpg - 259 KB