Autodesk®
Scaleform®

HUD 工具箱概述

本文介绍 Scaleform 4.2 HUD 工具箱。这是一个针对第一人称射击游戏的全功能、AAA 级、可重复使用的用户界面解决方案。本文重点讲述 Flash UI 内容以及用来管理内容的 C++ 代码。

作者： Nate Mitchell, Prasad Silva
版本： 2.0
上次编辑时间： 2010 年 7 月 30 日

Autodesk®
GAMEWARE

Copyright Notice

Autodesk® Scaleform® 4.2

如何联系 Autodesk Scaleform：

文档	HUD 工具箱概述
地址	Autodesk Scaleform Corporation 6305 Ivy Lane, Suite 310 Greenbelt, MD 20770, USA
网站	www.scaleform.com
电子邮箱	info@scaleform.com
电话	(301) 446-3200
传真	(301) 446-3199

目录

1 引言

Scaleform HUD（抬头显示器）工具箱是一系列高性能、全功能、AAA 级质量的用户界面工具箱中的第一个。开发人员可以容易地自定义这些工具箱并将其拖动到游戏之中。HUD 工具箱演示如何使用新的 Scaleform® Direct Access API 来创建高性能的第一人称射击游戏 (FPS) UI。

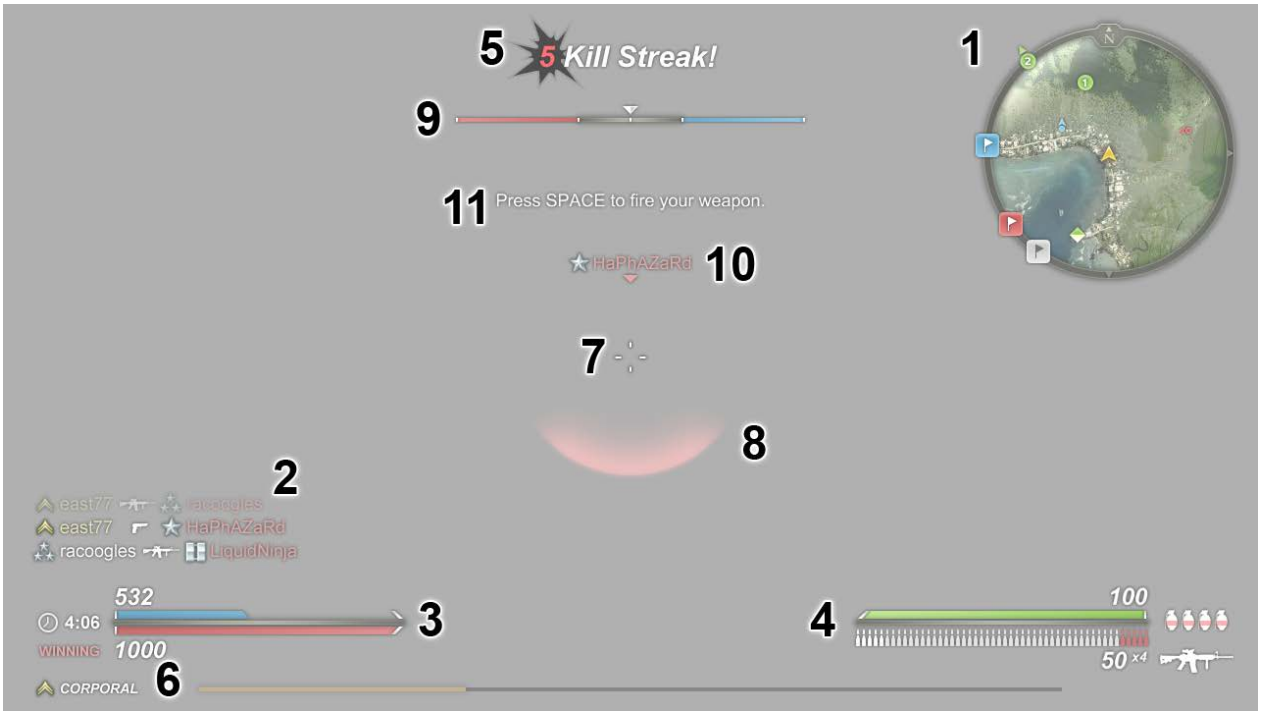

图 1：HUD 概览

HUD 工具箱包含下列可重复使用的 UI 元素：

1. 游戏迷你地图 (Minimap)
2. 游戏事件的动画日志
3. 记分牌和团队统计数据
4. 玩家健康 / 弹药 / 状态
5. 动画弹出式通知
6. 经验和级别显示
7. 动态十字线
8. 定向命中指示器
9. 目标捕获指示器
10. 广告牌 / 铭牌
11. 玩家文本通知

尽管此工具箱提供一套针对任何 FPS 游戏的开箱即用的解决方案，但用户不受已经提供的内容的限制。我们期望用户自定义或扩展此工具箱的各个元素，以为各种类型的游戏或应用程序创建新的具有创新意义的界面。

2 概述

2.1 文件位置和生成 (Build) 注释

与本演示关联的文件位于下列位置：

- *Apps\Kits\HUD* - 包含用于 HUD 演示可执行文件的 C++ 代码。
- *Bin\Data\AS2 or AS3\Kits\HUD* - 包含 Flash 资源和 ActionScript 代码（AS2 目录有 ActionScript 2、Flash 8 资源，然后 AS3 目录有 ActionScript 3、Flash 10 资源）。
- *Projects\Win32 \{Msvc80 or Msvc90}\Kits\HUD* – 包含用于 Windows 上运行的 Visual Studio 2005/2008 的演示项目。
- *Projects\Xbox360\{Msvc80 or Msvc90}\Kits\HUD* – 包含用于 Xbox 360 上运行的 Visual Studio 2005/2008 的演示项目。
- *Projects\Common\HudKit*.mk – 用于 HUD 工具箱的 PS3 生成文件。

Bin\Kits\HUD 中可以找到该演示（用于 Windows）的一个预先生成的可执行文件 HUDKit.exe。*它也可以经由开始菜单或 Scaleform SDK 浏览器进行访问。*

在 Windows 上，位于 *Projects\Win32\Msvc80\Kits*（或 *Msvc90\Kits*）目录的 Scaleform 4.2 Kits.sln 文件可用来生成和运行本演示。确保从解决方案运行本演示之前将用于 Debugging（调试）的“Working 目录”设置为 *Bin\Data\AS2\Kits\HUD or Bin\Data\AS3\Kits\HUD* 目录。

在 Xbox 360 上，位于 *Projects\Xbox360\{Msvc80* 或 *Msvc90}\Kits\HUD* 目录的 Scaleform 4.2 Kits.sln 文件可用来生成、部署和运行本演示。编译完成时，用于本演示的所有资源（位于 *Bin\Data\AS2 or AS3\Kits\HUD*）都将部署到目标 Xbox 360。

对于 PS3，应从 Scaleform 安装目录的根部运行 *make* 命令。默认情况下，这将会生成包括 HUD 工具箱 (HUD Kit) 在内的所有可用演示。在 PS3 上，可通过 SN Systems 工具集启动可执行文件。.该可执行文件将生成到 *Bin\PS3* 并应使用以下选项进行启动：

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

2.2 演示使用

为了最好地演示 HUD 工具箱的使用方法，该项目包含有一个非常简单的模拟游戏，其中包含两个队：红队和蓝队，两队为控制散布在战场上的一组目标而展开竞赛。

分数是通过捕获目标（迷你地图上标出）和击毙敌方玩家来产生的。捕获目标每 3 秒钟就会为目前控制该目标的队产生 1 分。击毙一个敌人值 1 分。累计获得 500 分的第一个队，或一个回合（15 分钟）终止时拥有最高得分的队，将被宣布为获胜者。

玩家开始时配备有一挺机关枪、一把手枪、4 枚手雷和充足的弹药。如果一个玩家被击毙，其武器和弹药将会得到补充。

以旋转的绿色钻石表示的能力提升 (Power-up) 将在战场上大量产生随机位置。每次能力提升都会对玩家的健康和拾取的随机武器的弹药产生预定的增强。

2.3 控制方案

2.3.1 键盘 (Windows)

W, A, S, D	控制用户玩家。WS 可将玩家向前和向后移动。A, D 可分别将玩家向左和向右移动。
SPACE	用当前武器射击。 所配备武器的射程、损伤能力和射速是独一无二的。
R	给当前武器填弹。
G	投手雷。将对准射程内最近的玩家并爆炸，对爆炸半径内的所有敌人造成损伤。
ESC	在标题栏与文本框 (Text Field) 之间切换，可显示应用程序的显示器统计数据。
B	切换对用户玩家的 AI 控制。禁用游戏输入，直至失效。
1, 2	更换武器。 • 1 选择手枪。 • 2 选择步枪。

2.3.2 控制器 (Xbox/PS3)

定向垫 **左控制杆**	控制用户玩家。W, S 可将玩家向前和向后移动。A, D 可分别将玩家向左和向右移动。
右触发器	用当前武器射击。 所配备武器的射程、损伤能力和射速是独一无二的。
X / 方块	给当前武器填弹。
左触发器	投手雷。将对准射程内最近的玩家并爆炸，对爆炸半径内的所有敌人造成损伤。
B / 圆圈	在标题栏与文本框之间切换，可显示应用程序的显示器统计数据。
Y / 三角形	循环切换武器。

3 架构

HUD 工具箱包含 Flash UI 资源以及更新 HUD 视觉状态的 C++ 代码。同时还有一个 C++ 模拟，提供一个以虚拟数据驱动 HUD 的环境。

3.1 C++

C++ 代码的两个主要部分是连接 HUD 视图的接口以及向该视图提供数据的环境。在 MVC 范例中，接口就是控制器，环境为模型，而 Flash UI 是视图。带 Fx 前缀的文件和类定义 HUD 工具箱的“视图” (View) 核心的接口。没有此前缀的文件定义本演示或模拟游戏的接口。

适配器设计方案的实现目的是使模拟与 HUD 视图脱钩。适配器翻译模拟与 HUD 视图之间的请求。要想用接口连接 HUD 视图与一个新游戏，用户需要开发一个自定义的适配器类，该类应有连接游戏与 HUD 视图的接口。这样，开发者就可以方便地重复使用 HUD 工具箱，而不必大幅度地修改游戏代码。

本文详述的代码介绍一种 Scaleform 的优化实现方法，它利用新的 Scaleform Direct Access API 来比以前更快地实现 UI 更新和动画。

C++ 代码包含下述文件（位于 *Apps\Kits\HUD* 及其子文件夹）。

3.1.1 演示

- **HUDKitDemo.cpp** – 在标准 Scaleform Player 基础上构建的核心应用程序。处理启动模拟、初始化成员以及利用其 C++ 控制器加载和注册 SWF 文件的工作。
- **HUDKitDemoConfig.** – 针对基于 FxPlayerConfig.h 的 HUDKitDemo 的配置。包括控制器映射和 Windows 应用程序定义。

3.1.2 HUD/迷你地图视图核心

- **FxHUDKit.h** – 用来更新 HUD 视图的核心 HUD 工具箱类型。
- **FxHUDView.h/.cpp** – 提供 HUD 视图使用的类型，包含日志和广告牌 (Billboard) 系统。
- **FxMinimap.h** – 声明游戏环境和应用程序为插到迷你地图视图中而可实现的接口。
- **FxMinimapView.h/.cpp** – 提供迷你地图视图使用的类型。

3.1.3 模拟游戏

- **HUDAdapter.h/.cpp** – 实现适配器设计方案，使模拟与 HUD 脱钩，以便于重复使用。
- **HUDEntityController.h/.cpp** – 声明针对实体的控制器类型，主要用于更新模拟逻辑和 AI 行为。
- **HUDSimulation.h/.cpp** – 提供实现演示环境以驱动游戏 UI 的类型。此环境包含属于一种捕获并占有游戏类型的两个队的玩家。

3.2 Flash

用于 HUD 工具箱的 Flash 内容分为两个文件：**HUDKit.fla** 和 **Minimap.fla**。这两个 FLA 文件定义和布置 HUD 工具箱的 UI 元素，以便于用 Scaleform 进行操纵。它们还包括 UI 中显示的所有图像和图标。

两个文件都分成若干层。一般来说，每个组件或每个具有相似组件的组都拥有自己的层。层提供一种方便的方法，用来在创作时根据深度控制元素排序。最上面的一层将显示在所有其他层的上面，以此类推。

在 Scaleform 中，可使用 C++、ActionScript 或 Flash 时间轴来处理 Flash 动画。在本演示中，多数 HUD 动画是通过 Classic Tweens 在 Flash 时间轴上处理的。这些动画一般通过调用 GFx::Value::GotoAndPlay() 来从 C++ 触发。

3.2.1 HUDKit.fla

位于 *Bin\Data\AS2 or AS3\Kits\HUD* 目录的 HUDKit.fla 文件是本演示的主要 SWF。此文件由 HUDKitDemo 应用程序在运行时加载。每个 Flash UI 元素都存在于此文件，但 HUDKit.fla 在运行时加载的迷你地图视图除外。

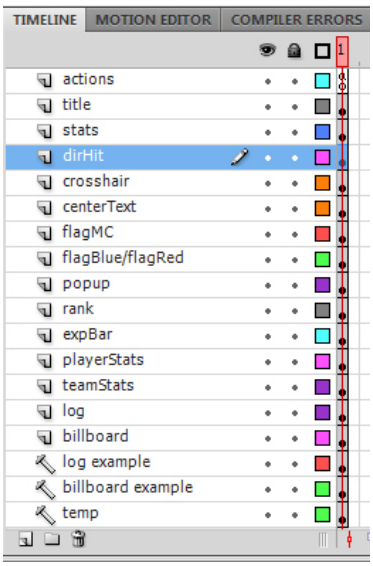

图 2：HUDKit.fla 层

HUDKitDemo 通过在时间轴的 Frame 1（帧 1）上调用 External Interface 来注册 HUDKit MovieClip：

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

这会传递一个指向该 MovieClip（电影剪辑）的指针，C++ 可将该 MovieClip 注册到 C++ HUDView 以便于操纵用户界面。

本文“HUD 视图”一节中将详细讲述每个层、其 MovieClips、 布局、动画和 Scaleform C++ 管理器。

3.2.2 Minimap.fla


图 3：Minimap 符号

Minimap.fla 包括 'Minimap' 符号，即迷你地图的核心。就本演示的范围而言，背景是一个静态图像，而不是一个交互式 3D 环境。

玩家用位于中心的一个黄色箭头来表示，而标题显示在迷你地图边界上（“北”已标出）。除玩家之外，迷你地图视图还显示五个图标类型：

- *友方玩家（蓝队）*。方向箭头只出现在高缩放级别。
- *敌方玩家（红队）*。方向箭头只出现在高缩放级别。
- *捕获点*（用旗标 (Flag) 表示：白色代表中立，蓝色代表友方，而红色代表敌方）
- *能力提升*（旋转的绿色和白色图标）

捕获点和能力提升为粘图标，它们在视野范围外但在可探测范围内时维持在视图边界。 Direct Access API 方法用来提供使进入和离开可探测范围的图标淡入/淡出的功能。它还用来在运行时使用 C++ 附加 MovieClips 和从台 (Stage) 上删除 MovieClips。

此 'Minimap' 符号包含多个层。这些层提供一种方便的方法，用来在创作时根据深度控制元素排序。最上面的一层将显示在所有其他层的上面，以此类推。

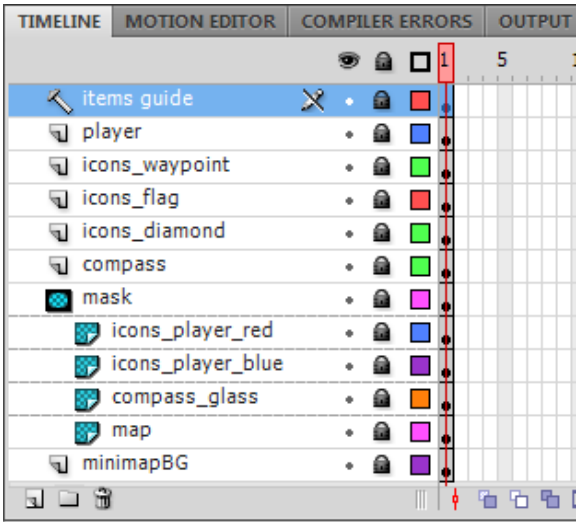

图 4：'minimap' 符号的层

- **items guide**：对于设计师/艺术家有用的引导层（Guide Layer）。此层的内容不发布到 SWF 文件。
- **player**：包含黄色玩家图标的玩家。
- **icons_waypoint**：生成并缓存航点图标的空白画布。
- **icons_flag**：生成并缓存旗标的空白画布。
- **icons_diamond**：生成并缓存能力提升的钻石图标的空白画布。
- **compass**：包含指南针的层（标有“北”）。
- **mask**：只显示未掩蔽区域的内容的掩蔽层 (Mask Layer)。
    - **icons_player_red**：生成并缓存红色敌方图标的空白画布。

    - **icons_player_blue**：生成并缓存蓝色友方图标的空白画布。
    - **compass_glass**：在指南针左上方提供玻璃光泽的装饰。
    - **map**：包含地形图的层。出于演示目的，我们对地形使用一个大型位图，不过，在制作情形下，这很可能是从 C++ 更新的纹理。在此情况下，更新逻辑不需要更改地图的偏移/旋转，这在使用掩蔽地形图时是必需的。
- **minimapBG**：包含与地形位图边缘恰当混合的一个蓝色背景。这在地形位图超出视野时提供一个平稳的过渡。如果地形图渲染为纹理并直接从 C++ 更新，就不需要该背景。

'Minimap' 符号使用 com.scaleform.Minimap 类（位于 *Bin\Data\AS2 or AS3\Kits\HUD\com\scaleform* 下面）。）此最小类定义地图图像的路径以及迷你地图的构造函数的路径，该构造函数将迷你地图注册到 HUDKitDemo 应用程序并加载地图图像。

4 HUD 视图

C++ HUD 视图管理内容 HUDKit.swf。其接口声明下列类型：

- FxHUDView – 用于 HUD（特别是 HUDKit.swf）的核心管理器 (Core Manager)。根据模拟状态更新视图和所有子 MovieClips。
- BillboardCache – 用于友方/敌方广告牌的管理器和缓存系统（中心、屏幕指示器，不要与迷你地图图标混淆）。
-
- FxHUDLog – 用于 HUD 左边显示的消息日志的管理器。跟踪、显示和重复使用 FxHUDMessages。
- FxHUDMessage – 消息定义，包括电影剪辑引用和消息文本。

请注意，FxHUDView 的缓存电影剪辑引用是用 *MC*（MovieClip 的简写）后缀表示的。

在其初始化期间，FxHUDView 将一个引用注册到运行时在 UpdateView() 中更改的每个主要 MovieClip。这对于避免在每次更新时都检索电影剪辑引用非常重要，因而可以最大限度地减少用于更新视图的时间。初始化还隐藏 HUDKit.fla 中定义的许多未使用的 UI 元素。

FxHUDView::UpdateView() 用模拟环境提供的数据更新视图的每个元素。其逻辑分为以下几个子方法：UpdateTeamStats()、UpdatePlayerEXP()、Update PlayerStats()、UpdateEvents()、UpdateTutorial() 和 UpdateBillboards()。一个指向模拟环境的指针会被传递到其中每个方法，该模拟环境然后用来更新特定 UI 元素。

请注意，很少在调用 UpdateView() 期间检索电影剪辑引用。尽可能避免不必要的电影剪辑更新，因为每个帧都更新电影剪辑是一种潜在的循环动作浪费。

有了该版本 Scaleform Direct Access API，现在可以在 Flash 与应用程序之间高效地传递任何类型和形式的数据。尤其是，GFx::Value 类提供许多旨在使 Scaleform 用户能够更有效地控制 Flash 内容的新函数。GFx::Value::SetDisplayInfo()、GFx::Value::GetDisplayInfo() 和 GFx::Value::SetText() 用于整个 HUD 视图的更新逻辑之中，便于直接而高效地操纵电影剪辑的显示状态。

4.1 回合统计数据和记分牌

teamStats MovieClip（位于 HUDKit.fla 的场景 1 (Scene 1) 中的 teamStats 层）是进行中的回合的记分牌和统计数据。它包括胜负文本框 (*redWinning*, *blueWinning*)、两个文本框 (*scoreRed*, *scoreBlue*) 中显示的各队的得分以及进度条 (*teamRed, teamBlue)*，还有回合时间时钟文本框 (*roundTime*)。

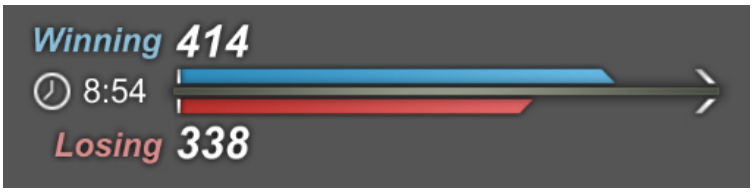

图 5：teamStats 符号

专门使用 FxHUDView::UpdateTeamStats() 来更新 *teamStats*。下面的代码更新 *teamStats* MovieClip 的元素。

```
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;
      }
}
```

得分存储在一个 Scaleform::String 中，并使用 GFx::Value::SetText() 传递到 *scoreRed* 和 *scoreBlue* 文本框。teamRed 和 teamBlue 得分条的 X 标尺是使用 GFx::Value::SetDisplayInfo() 进行更新的。二者从其 0% 原始宽度开始，随着各队的得分向赢得相应回合所需的得分增加而逐渐伸展。

更新 *blueWinning/redWinning* 文本框之前，对比两队得分以免进行不必要的更新。调用 GFx::Value::SetText() 以便使用 HUDView 初始化期间定义的恒定 “Winning” 字符串和 “Losing” 字符串来设置文本。更新回合时间 textField 时，将最后一次调用 UpdateView() 时剩余的时间与最新时间进行对比。如果上次更新以来过去的时间已经超过一秒，则使用 GFx::Value::SetText() 并通过格式化的时间字符串来更新 *roundTime* textField。

4.2 Player Stats 和 Ammo

playerStats 符号（位于 HUDKit.fla 的场景 1 中的 playerStats 层）包含健康文本框和健康条形状 (*healthN, health)*、武器图标符号 (*weapon*)、HUD 的弹药指示器以及弹药不足指示器 (*reload*)。

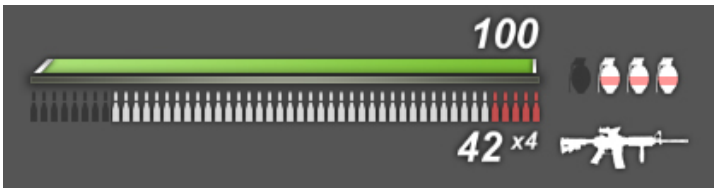

图 6：playerStats 符号

弹药指示器分为六个电影剪辑，它们是 *clipN*、*ammoN*、*ammo*、*ammoBG*、*grenade* 和 *grenadeBG*。*文本框 clipN 和 ammoN* 以武器的弹夹和装填的弹药的形式显示剩余弹药的数量。

ammo/healthVehicle 层包含两个符号：*ammo* 和 *ammoBG*，每个符号分别被分隔到一个适当的子层中。两个符号都在健康条下包含有弹药指示器形状。每个符号都分为 51 个 Keyframes（关键帧），每个关键帧拥有的子弹指示器都比前一个关键帧少一个。GFx::Value::GotoAndStop() 用来访问这些电影剪辑的不同的关键帧，更改显示的形状的数量，以反映模拟的状态。

- *ammo* 显示所配备武器中已装填的子弹的数量。
- *ammoBG* 显示所配备武器的弹夹中的最大子弹数量。

grenade/armor 层与 ammo 层的设置相似，也是有两个符号：*grenade 和 grenadeBG*，每个符号被分隔到各自的层中。两个符号都包含手雷指示器形状。就像在弹药指示器中一样，其中每个符号都分成 5 个关键帧。帧 1 有 4 个手雷形状，而帧 5 没有手雷形状。GFx::Value::GotoAndStop() 用来访问这些电影剪辑的不同的关键帧，更改显示的形状的数量，以反映模拟的状态。

- *grenade* 显示剩余的手雷的数量。
- *grenadeBG* 显示玩家可以携带的手雷的最大数量。

下面的逻辑更新 *playerStats* 符号。

```
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;
	}
}
```

在 *ammo* 中，帧 1 有 50 个子弹指示器，而帧 51 有 0 个子弹指示器。每次使其武器射击时，都会调用 ammoMC.GotoAndStop(ammoFrames - ammoInClip) 以便访问适当的关键帧，从 HUD 中删除一个已装填的子弹指示器。

对于 *ammoBG*，使用 ammoBGMC.GotoAndStop() 来访问显示 X 个子弹指示器背景的关键帧。这里，X 基于所配备武器的弹夹大小。例如，机关枪 (Machine Gun) 的弹夹包含 50 发子弹。因此，ammoBG.GotoAndStop(1) 将显示帧 1 中显示的所有 50 个子弹指示器背景。只有在上次调用 FxHUDView::UpdateView() 以来所配备武器发生变化的情况下才更新此 MovieClip。

4.3 旗标捕获指示器

flagMC 符号（位于 HUDKit.fla 的场景 1 的旗标层）是旗标捕获指示器的容器。*flagMC* 包含 *flag* 符号，该符号分为两部分：旗标捕获指示器位图和箭头指示器 (*arrow)*，后者根据旗标的 CaptureState（捕获状态）向左和向右移动。当玩家接近一个捕获目标时，该指示器淡入，而当玩家不再射程内时，该指示器将淡出。

图 7：flagMC 符号

用于 *flagMC* 的淡入和淡出动画是使用一个 Classic Tween 在 Flash 时间轴上完成的。*flagMC* 的时间轴有 4 个图形状态，它们分别用 4 个关键帧表示：Hidden（隐藏）、Visible（可见）、Fade-In（淡入）和 FadeOut（淡出）。这些状态是使用 GotoAndStop() 和 Keyframe 的标签/编号（1、5、“On”、“Off” 等）访问的。

下面复制的 FxHUDView::UpdateEvents() 使用本模拟的数据来控制 *flagMC* 的状态并更新 *arrow*（箭头）的位置。

```
// 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;
      }
}
```

当调用 flagMC.GotoAndPlay("on") 时，*flagMC* 播放其淡入动画，然后停在 Visible（可见）状态。当调用 flagMC.GotoAndPlay("off") 时，*flagMC* 播放帧 11-20，即一个淡出动画。当动画达到帧 20 时，会自动重新启动从帧 1 回放。这是默认 Flash 行为；不需要额外的 ActionScript 或 C++。由于帧 1 包含一个 `stop();` 调用，因此，电影剪辑将在播放帧 1 后停止。这是非常理想的情况，因为符号隐藏在帧 1 中，这是由于其 Alpha 设置为 0。

arrow 是使用 GFx::Value::SetDisplayInfo() 横向转移的。此方法被传递来一个新的 GFx::Value::DisplayInfo，带有根据实体的 CaptureState 的更新的 X 坐标。

4.4 武器十字线

Reticule（十字线）符号（位于 HUDKit.fla 的场景 1 的准星层）是射击十字线的容器。*reticule* 包含四个符号：*left*、*right*、*bottom* 和 *top*，每个符号都在自己的层内。自己的符号内拥有每个十字线使得单独操纵成为可能。十字线与每当用户玩家用其武器射击时模拟所启动的 PlayerFire 事件相对应。当玩家用武器射击时，十字线将会动态扩大。当玩家停止射击时，十字线将会返回到其原始位置。

图 8：reticule 符号

十字线更新代码分为两个部分：FxHUDView::UpdateEvents() 方法和 FxHUDView::OnEvent() 方法。当捕获到一个 PlayerFire 事件时，ReticuleHeat 和 ReticuleFiringOffset 在 FxHUDEvent 中分别递增并设定。

```
// 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() 使用这两个变量来切换 *top*、*bottom*、*left* 和 *right* 电影剪辑。对于每个电影剪辑，我们使用 SetDisplayInfo() 来切换电影剪辑的 X 坐标或 Y 坐标。方程中的第一个数字（6 或 -6）是从 0, 0 坐标偏移的起始 X 或 Y。请注意，可以针对每次调用 SetDisplayInfo() 重新使用 GFx::Value::DisplayInfo 的单个实例。不过，为了确保不会发生意外重新使用 DisplayInfo 的情况，在每次调用 GFx::Value::SetDisplayInfo() 后再调用 DisplayInfo.Clear()。此方法清除了用于 DisplayInfo 实例的所有以前定义的显示属性。

```
// 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();
}
```

对于本演示，为了简单起见，选择了一个简单的动态十字线方程，但此代码可以容易地修改，以创建更加动态和有创意的十字线行为。

4.5 定向命中指示器

dirHit 符号（位于 HUDKit.fla 的场景 1 的 dirHit 层）是定向命中指示器的容器。*dirHit* 包含八个电影剪辑，它们被分成适当命名的层：*tl*、*l*、*bl*、*b*、*br*、*r*、*tr* 和 *t*。

其中每个电影剪辑都包含一个 Alpha 混合红半圆，将作为一个指示器用于：

a) 受到损伤的用户玩家
b) 该损伤的来源的相应方向

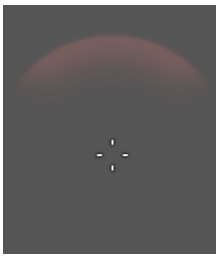

图 9：dirHit 符号

当一个玩家被命中时，就会出现命中指示器，随后在 ~1 秒钟后淡出。淡出动画是 Flash 时间轴上的一个 Classic Tween Alpha 混合。通过针对任何命中指示器电影剪辑调用 GFx::Value::GotoAndPlay(“on”)，相应命中指示器将会显示，过一秒钟后淡出。相应的电影剪辑基于损伤来源的相应方向。

```
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 等级和经验显示

expBar 符号（位于 HUDKit.fla 的 Scene 1 的 expBar 层）是屏幕底部的经验条，通过玩家的当前等级来跟踪玩家的进度。*expBar* 包含 *exp* MovieClip，后者是随玩家通过击毙和捕获来获得经验而扩大的黄色进度条。

rank 符号（位于 HUDKit.fla 的 Scene 1 的 rank 层）显示玩家的当前等级。*rank* 包含 18 个 Keyframe，每个专用于一个等级。每个 Keyframe 显示与该等级关联的图标和标题。

图 10：expBar 和 rank 符号

```
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);  // 伸缩 EXP 进度条
                                                              // 电影剪辑。
            ExpBarMC.SetDisplayInfo(info);
      }
}
```

与团队得分进度条相似，当玩家的经验发生变化时，*exp* 也根据玩家的当前经验而伸缩。为了更新 *rank*，将 GFx::Value::GotoAndStop() 用于关键帧 X，其中 X 为玩家的当前等级 + 1（因为没有 帧 0）。

4.7 文本通知

centerTextMC 符号（位于 HUDKit.fla 的场景 1 的 centerText 层）是一个动画式文本框，用来向屏幕中心的用户显示信息。在本演示中，*centerText* 用来在启动各个回合时显示一个简单的教程，并宣布该用户玩家最近已被击毙。该文本显示后随即淡出。

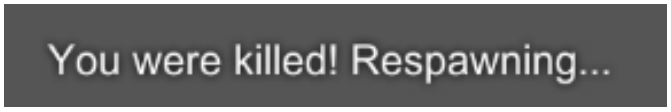

图 5：centerTextMC 符号

centerTextMC 包含 *centerText* 电影剪辑，后者包含实际的文本框 *textField*。如此排列电影剪辑的目的是便于制作动画。*centerTextMC* 的时间轴包含两个带有标签的关键帧 “on5”（在播放一个淡出动画前显示文本 5 秒钟）以及 “on”（在播放一个淡出动画之前显示文本 3 秒钟）。时间轴的末尾重新启动电影剪辑在帧 1 回放，在这里，*centerTextMC* 的 Alpha 设置为 0，将其隐藏直到重新调用 “on” 或 “on5”。

```
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.
```

在此，对 *centerTextMC* 的 *textField* 进行了适当设置，并使用 GotoAndPlay(“on”) 显示电影剪辑。电影剪辑将会由于时间轴的 Classic Tween 而在 ~3 秒钟后自行淡出。

4.8 弹出消息和通知

popup 符号（位于 HUDKit.fla 的场景 1 的 popup（弹出消息）层）是动画式事件指示器。在本演示中，每当用户玩家达到 3 次以上连续击毙（连杀）时，就会播放 *popup*。它将淡入并在显示时播放一个动画，而在 3-5 秒钟后根据 C++ 调用而淡出。

图 6：popup 符号

Popup 是弹出消息的得分容器。其子元素的所有动画都是使用 Classic Tweens 在 *popup* 的时间轴上完成的。*popup* 包含 4 层：*actions*、*popupNumber*、*popupText* 和 *popupBG*。*popupBG* 是在数字后面 “爆炸” 的动画黑色形状。*popupText* 和 popupNumber 包含显示弹出消息的文本和次数的 *textField* 符号。此符号层次是在时间轴上制作文本框动画所必需的。

此符号有一个 “on” 关键帧，可以使用 GotoAndPlay() 来调用该关键帧以启动动画。一旦完成初始动画，弹出消息就会淡出，而且回放将返回到帧 1 并停止播放，通过将其 Alpha 设置为 0 隐藏起来。

```
// 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); // 设置 PopUp 的文本框。
```

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

这里，FxHUDEvent 作为一个 KillStreak 事件被丢弃。然后从玩家检索连续击毙的次数并相应地格式化。通过 C++ 设置预缓存的 *popupText*'s *textField* 符号 PTextFieldMC 的文本，而 GotoAndPlay("on") 用来触发 *popup* 的显示动画。

4.9 消息和事件日志

log 符号（位于 HUDKit.fla 的 Scene 1 的 log（日志）层）是 HUD 左下边的消息和事件日志的容器。在本演示中，日志显示与最近事件（包括击毙、能力提升、旗标捕获和级别提升）相关的文本消息。日志底部添加新消息，就会向上推送旧消息。日志消息在显示 ~3 秒钟后淡出。

图 13：log 和 logMessage 符号

日志消息是 *logMessage* 符号的实例。在淡出动画开始之前，添加到日志的消息显示大约 3 秒钟。此动画是在 *logMessage* 符号的时间轴上定义的。消息的文本从 C++ 发送，并使用 htmlText 在 textField（文本框）中显示。

下面的代码来自 `FxHUDMessage::Movieclip* FxHUDLog::GetUnusedMessageMovieclip()`，后者创建一个新的 LogMessage MovieClip 并将其附加到 Log 日志电影剪辑，即用于所有日志消息的画布。

```
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;
}
```

消息的内容从 C++ 按事件类型设置。下面是 C++ 代码，用来处理 Level Up 事件，并为将在日志中显示的 “You are now a [Rank Icon] [Rank Name]”（您现在是 [等级图标] [等级名称]）新消息设置 htmlText。

```
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 秒钟
Log.PushBack(m);
NumMessages++;
```

下面是用来管理日志的 HUD Update 逻辑。它更新用于日志消息引用的布局和动画。

```
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 的更新逻辑通过检查某个 FxHUDMessage 是否设置为 0 来确定日志中目前是否显示该 FxHUDMessage。如果没有显示，就将其从 Log 中删除，并添加回到未用 FxHUDMessage MovieClips 的 MessageMCs 列表。如果添加了一个新的 FxHUDMessage，就会使用 GFx::Value::DisplayInfo.SetY() 和 GFx::Value::SetDisplayInfo() 将所有 MessageMC 相应升档。

4.10 广告牌

billboard_container（位于 HUDKit.fla 的 Scene 1 的 billboard 层）是玩家广告牌的容器。每个广告牌显示一个跟随玩家位置的箭头和玩家名称。广告牌经常在游戏中显示 3D 对象的附加详细信息，而 3D 对

象包含在用户玩家的视图平截体 (Frustum) 内。在本演示中，玩家广告牌出现在用户玩家的一定距离处。友方玩家始终显示玩家名称，而敌方广告牌只显示箭头，直到目标接近视图中心。

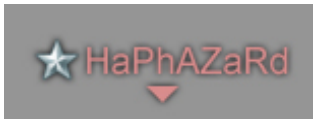

图 7：billboard_enemy 符号

本演示中的广告牌实现在 UI 中创建了一个 2D 广告牌电影剪辑，通过切换和伸缩并使用 GFx::Value::SetDisplayInfo() 来跟踪玩家的位置。

每个广告牌 MovieClip (*billboard_friendly, billboard_enemy)* 都是从位于文本层的一个 *billboard_label_(friendly/enemy)* 以及位于箭头层的一个箭头图像构造的。*billboard_label_** 包含一个用来显示目标玩家名称的 textField。*billboard_friendly*'s "Show" Keyframe 使用 *billboard_friendly* 的时间轴上的 Classic Tween 半透明混合在启动淡入动画。一旦播放 "Show"（显示），广告牌就保持可见，直到其被隐藏或删除为止。

将广告牌电影剪辑附加到画布的 C++ 逻辑在 `FxHUDView::BillboardCache::GetUnusedBillboardMovieclip` 内：

```
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);

}
```

每个 BillboardCache 都包含一个对电影剪辑的引用，它将电影剪辑附加到广告牌命名的 CanvasMC。初始化时，此引用被传递到 BillboardCache。上面的逻辑将 billboard_enemy/billboard_friendly 符号的一

个广告牌电影剪辑附加到其位于 CanvasMC.AttachMovie() 调用的预订义画布，并保留一个对它的引用，以便于重新使用。

Billboard 更新逻辑是在下列方法中定义的：UpdateBillboards()、BeginProcessing()、EndProcessing() 和 GetUnusedBillboardMovieclip()。在更新一组友方广告牌或敌方广告牌的开始和结尾调用 BeginProcessing() 和 EndProcessing()。这些方法管理已使用的和未使用的广告牌列表。

下面的代码是来自 UpdateBillboards() 的友方广告牌的更新逻辑。

```
// 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() 根据一个从用户玩家视图查询检索到的实体列表创建和更新广告牌。它使用 GotoAndPlay(“show”) 显示广告牌，使用 GFx::Value::SetDisplayInfo() 设置 MovieClip’s X、Y 和标尺，并使用 GFx::Value::SetText() 更改显示的文本。

textField 使用 htmlText 填充和显示广告牌。该文本框包含玩家等级图像和玩家名称。等级图标通过 *src='rank{0}'* HTML 定义。在上面的示例中，它与友方玩家等级 + 1 (`friendlies[i].pPlayer->GetRank()+1`) 相对应。其他 *img* 选项定义该图标图像的高度、宽度、对齐方式和 vspace。玩家的名称通过 {1} 定义。在上面的示例中，它与友方玩家的名称 (`friendlies[i].pPlayer->GetName()`) 相对应。

5 迷你地图视图

迷你地图（Minimap) 的 C++ 方面包含以下文件（位于 *Apps\Kits\HUD* 之中）：

- **FxMinimap.h**：声明游戏环境和应用程序为插到迷你地图演示核心中而可以实现的接口。
- **FxMinimapView.h/.cpp**：提供迷你地图视图使用的类型。

5.1.1 迷你地图视图

迷你地图视图接口声明下列类型：

- **FxMinimapIconType**：图标类型定义，包括类型 ID 以及用于额外专业化的额外子类型值。
- **FxMinimapIcon**：与视图中某个元素链接的图标资源。
- **FxMinimapIconCache**：存储特定类型图标集的图标缓存。根据需要由该缓存来管理图标视图状态，而且该缓存还支持进入/退出可探测范围的图标的淡入/淡出。该缓存使用一个工厂来生成新图标。工厂的基实现是一个模板化的类，称为 MinimapIconFactoryBase。各个工厂处理根据需要把图标的电影剪辑附加到其适当画布电影剪辑以及在迷你地图被毁坏时删除这些 MovieClips 方面的事情。定义了多个工厂类型来支持下列不同的图标类型。各个图标类型都包含一个 Update() 方法，该方法实现针对图标类型的更新逻辑，例如：翻译、旋转和 textField 更改：
    - *PlayerIcon*：同时代表友方和敌方图标资源，因为两种图标类型共享相同的逻辑。它们只在图标创建方法方面存在区别。
    - *FlagIcon*
    - *ObjectiveIcon*
    - *WaypointIcon*
- **FxMinimapView**：迷你地图视图的主控制器。应用程序调用其 UpdateView() 方法来刷新视图。

下面的代码是 PlayerIcon::Update() 方法（FxMinimapView.cpp；第 116 行）。它显示用来利用 Direct Access API 更新视图中的玩家图标的逻辑。*MovieClip* 成员包含对台上的图标 MovieClip 的引用。SetDisplayInfo 方法直接修改电影剪辑的显示属性（显示矩阵、可视性旗标或 Alpha 值），而不是通过较慢的 SetMember() 方法。请注意，GFx::Value::SetMember 仍然比 GFx::Movie::SetVariable 快：

```
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 = (int)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 性能分析

应用程序显示的统计数据说明更新 HUD 视图所花费的时间、更新迷你地图所花费的时间、已更新的迷你地图对象的数量、Scaleform 内容的总显示时间以及推进 SWF 所花费的时间。

对象数量是各种类型的实体（包括友方/敌方玩家、能力提升和目标）的累计数量。HUD 的更新时间包括更新所有 UI 指示器，例如：广告牌、日志消息、回合记分牌以及玩家统计数据。迷你地图的更新时间包括更新前面提到的迷你地图对象以及掩蔽地形、指南针和玩家图标所花费的时间。更新这些视图包括执行控制电影剪辑（如 x/y 位置、标签、当前帧、旋转、缩放和转换矩阵）视觉属性的所有逻辑。

6.1 性能统计数据

以前的 Scaleform 实现仅限于使用 C++ GFx::Movie::SetVariable 和 GFx::Movie::Invoke 方法更新 Movie 的视图状态。令人遗憾的是，由于许多因素（如解析 MovieClip 路径）所致，这产生极大的性能损失。相比之下，Direct Access API 提供便于执行同样的功能的更快的路径，因此，使得使用 Scaleform 3.1 和更新版本进行复杂 HUD 更新高效得多。

下图显示采用 Direct Access API 方法与采用 SetVariable/Invoke 方法相比迷你地图性能方面的显著增强（缩短了更新时间）。性能增强为 10-25 倍：

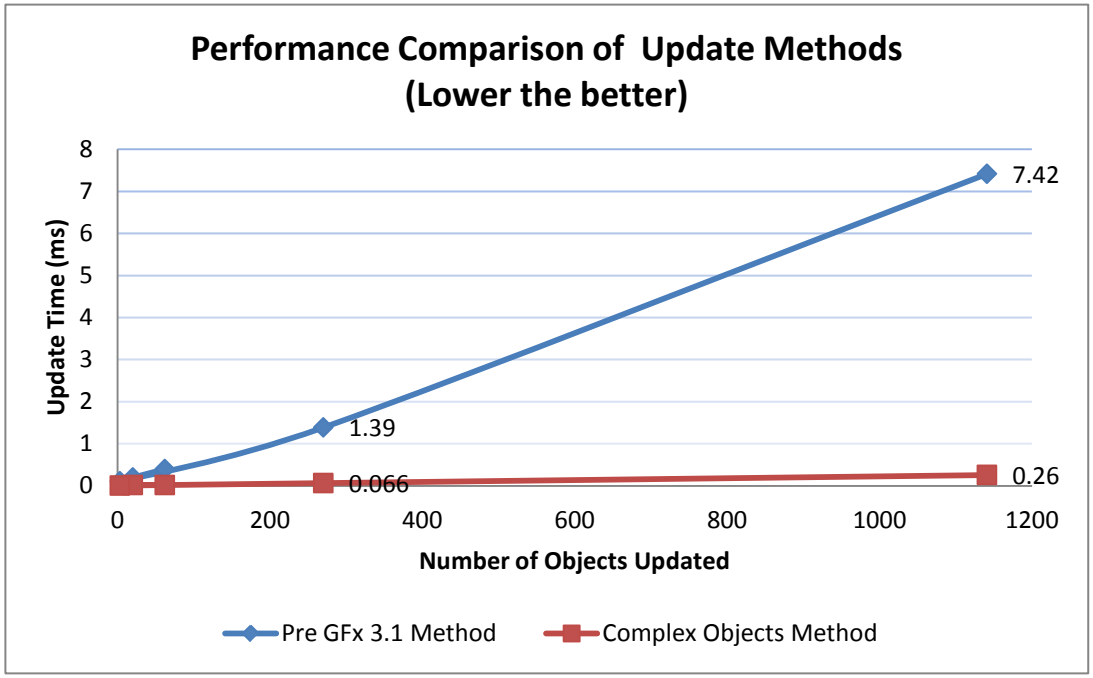

图 8：Direct Access API 与 SetVariable/Invoke 之间的迷你地图更新方法性能比较

如上图所示，Direct Access 方法的更新时间对于各种工作环境来说都是最佳的，因为更新时间远远低于 1 毫秒的阈值 – HUD UI 的分配的平均处理时间。

下面按平台显示一次 15 分钟模拟回合期间整个 HUD 工具箱的一般性能统计数据。这些数字提供 Scaleform 驱动的全功能 Flash HUD 的性能估计：

平台	FPS	更新（毫秒）	显示（毫秒）	推进（毫秒）	总计（毫秒）
Windows Vista 在 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：Scaleform HUD 工具箱每个平台的平均性能统计数据

如表 1 所示，HUD 的更新时间对于各种工作环境来说都是最佳的，因为更新时间远远低于 1 毫秒的阈值 – HUD UI 的分配的平均处理时间。

6.2 内存明细表

下面是 HUD 工具箱演示加载的所有未压缩内容的一个内存明细表。内容是使用 GFxExport 的默认设置导出的。全部内存占用包括 .GFX 文件、图像、组件和导出的字体。

1. **文件格式：　GFX 标准导出、未压缩的图像**

总内存：　1858kb

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

未压缩的图像

- HUDKit.gfx
  - 核心 UI 组件　- 603kb
  - 等级图标　- 72kb
  - 武器图标　- 51kb
    - **注意：**本版 HUD 工具箱中，有 43kb 的武器图标没有使用。
- Minimap.gfx
  - 组件　- 644kb
- Map.jpg　- 259kb