Autodesk®
Scaleform®

HUD Kit Overview

This document describes the Scaleform 4.3 HUD Kit, a fully-featured, AAA, reusable user interface solution for a first-person shooter game. It focuses on the Flash UI content and C++ code used to manage it.

Author: Nate Mitchell, Prasad Silva

Version: 2.00
Last Edited: July 30, 2010

Autodesk®
GAMEWARE

Copyright Notice

Autodesk® Scaleform® 4.3

How to Contact Autodesk Scaleform:

Document	HUD Kit Overview
Address	Autodesk Scaleform Corporation 6305 Ivy Lane, Suite 310 Greenbelt, MD 20770, USA
Website	www.scaleform.com
Email	info@scaleform.com
Direct	(301) 446-3200
Fax	(301) 446-3199

Table of Contents

1 Introduction

The Scaleform HUD (Heads Up Display) Kit is the first in a series of high performance, full-featured, AAA quality user interface kits which developers can easily customize and drop into their game. The HUD kit demonstrates how to create a high performance first-person shooter (FPS) UI using the Autodesk® Scaleform® Direct Access API.

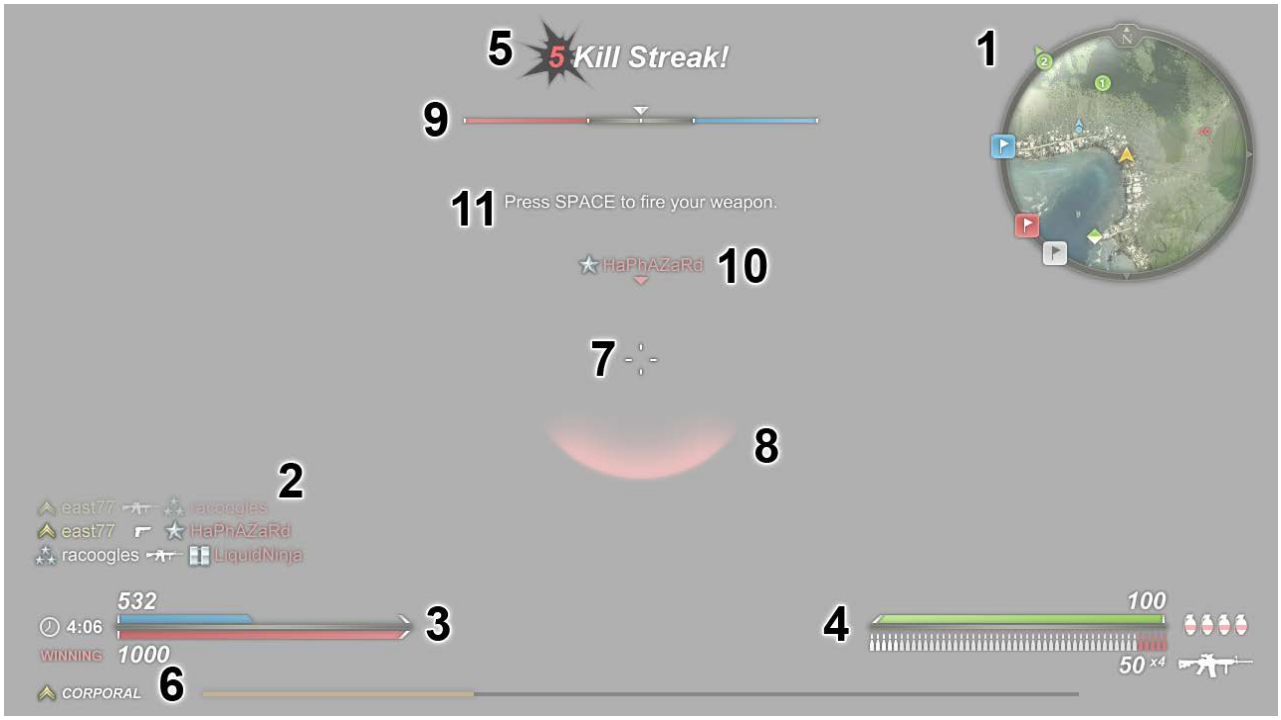

Figure 1: HUD Overview

The HUD kit features the following reusable UI elements:

1. Game Minimap
2. Animated Log of Game Events
3. Scoreboard and Team Statistics
4. Player Health / Ammo / Status
5. Animated Pop-Up Notifications
6. Experience and Rank Display
7. Dynamic Reticule
8. Directional Hit Indicator
9. Objective Capture Indicators
10. Billboards / Nameplates
11. Player Text Notifications

Although this kit provides an out of the box UI solution for any FPS game, users are not limited by the content provided. We expect users to customize or extend individual elements of this kit to create new and innovative interfaces for any type of game or application.

2 Overview

2.1 File Locations and Build Notes

The files associated with this demo are located in the following locations:

- *Apps\Kits\HUD* - Contains C++ code for the HUD Demo executable.
- *Bin\Data\AS2 or AS3\Kits\HUD* - Contains Flash assets and ActionScript code (AS2 directory contains ActionScript 2/Flash 8 assets, and AS3 directory contains ActionScript 3/Flash 10 assets).
- *Projects\Win32 \{Msvc80 or Msvc90}\Kits\HUD* – Contains the demo projects for Visual Studio 2005/2008 on Windows.
- *Projects\Xbox360\{Msvc80 or Msvc90}\Kits\HUD* – Contains the demo projects for Visual Studio 2005/2008 on Xbox 360.
- *Projects\Common\HudKit*.mk – PS3 makefile for the HUD Kit

A pre-built executable of the demo for Windows, HUDKit.exe, can be found in the *Bin\Kits\HUD.* It is also accessible via the start menu or the Scaleform SDK Browser.

On Windows, the Scaleform 4.3 Kits.sln file located in the *Projects\Win32\Msvc80\Kits* (or *Msvc90\Kits*) directory can be used to build and run this demo. Be certain that the "Working directory" for Debugging is set to the *Bin\Data\AS2\Kits\HUD* or *Bin\Data\AS3\Kits\HUD* directory before running the demo from the solution.

On Xbox 360, the Scaleform 4.3 Kits.sln file located in the *Projects\Xbox360\{Msvc80 or Msvc90}\Kits\HUD* directory can be used to build, deploy, and run this demo. All of the assets for the demo, located in *Bin\Data\AS2 or AS3\Kits\HUD*, will be deployed to the target Xbox 360 when compilation completes.

For PS3, the *make* command should be run from the root of the Scaleform installation directory. This will, by default, build all available demos including the HUD Kit. On PS3, the executable can be launched via the SN Systems toolset. The executable will be built to *Bin\PS3* and should be launched using the following options:

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

2.2 Demo Usage

To best *demonstrate the use of the HUD Kit, the project contains* a very simple game simulation consisting of two teams, Red and Blue, vying for control over a set objectives spread across a battlefield.

Points are accrued by capturing objectives (marked on the minimap) and eliminating enemy players. A capture objective will generate 1 point every 3 seconds for the team who currently controls it. An enemy kill is worth 1 point. The first team to accumulate 500 points or the team with the highest score when the round time (15 minutes) expires is declared the victor.

Players begin equipped with a machine gun, a pistol, four grenades and full ammunition. If a player is killed, his weapons and ammunition will be replenished.

Power-ups, represented by spinning green diamonds, will spawn in random locations on battlefield. Each provides a predetermined boost to the player's health and ammunition for a random weapon when picked up.

2.3 Control Scheme

2.3.1 Keyboard (Windows)

W, A, S, D	Control the user player. W, S moves the player forward and backward. A, D moves the player left and right respectively.
SPACE	Fire current weapon. The range, damage, and rate of fire are unique to the weapon equipped.
R	Reload current weapon.
G	Throw a grenade. Will target closest player within range and explode, damaging all enemies within a radius of blast.
ESC	Toggles display of the application's display statistics between the title bar and text fields.
B	Toggles AI control of user's player. Disables gameplay input until deactivated.
1, 2	Change weapons. • 1 selects the pistol. • 2 selects the rifle.

2.3.2 Controller (Xbox/PS3)

D-Pad **Left Joystick**	Control the user player. W, S moves the player forward and backward. A, D moves the player left and right respectively.
Right Trigger	Fire current weapon. The range, damage, and rate of fire is unique to the weapon equipped.
X / Square	Reload current weapon.
Left Trigger	Throw a grenade. Will target closest player within range and explode, damaging all enemies within a radius of blast.
B / Circle	Toggles display of the application's display statistics between the title bar and text fields.
Y / Triangle	Cycle through weapons.

3 Architecture

The HUD kit consists of the Flash UI assets and the C++ code that updates the HUD's visual state. There is also a C++ simulation that provides an environment to drive the HUD with dummy data.

3.1 C++

The two main parts of the C++ code are the interface to the HUD view and the environment that provides data to the view. In the MVC paradigm, the interface would be the controller, the environment is the model and the Flash UI is the view. Files and classes that are prefixed with Fx define interfaces of the HUD Kit's View core. Files without this prefix define interfaces for the demo or game simulation.

The adapter design pattern is implemented to decouple the simulation from the HUD View. The adapter translates requests between the simulation and the HUD View. To interface the HUD View with a new game, users need to develop a custom adapter class which interfaces with their game and the HUD View. This allows developers to reuse the HUD Kit conveniently without significantly modifying their game code.

The code detailed within this document presents an optimal implementation of Scaleform, leveraging the new Scaleform Direct Access API to accomplish UI updates and animations faster than ever before.

The C++ code consists of the following files (located in *Apps\Kits\HUD* and its subfolders).

3.1.1 Demo

- **HUDKitDemo.cpp** – Core application built on top of the standard Scaleform Player. Handles starting the simulation, initializing members, and loading and registering the SWF files with their C++ controller.
- **HUDKitDemoConfig**. – Configuration for the HUDKitDemo based on FxPlayerConfig.h. Includes controller mappings and Windows application definitions.

3.1.2 HUD/Minimap View Core

- **FxHUDKit.h** – Core HUD kit types used to update the HUD view.
- **FxHUDView.h/.cpp** – Provides types used by the HUD view including the log and the billboard system.

- **FxMinimap.h** – Declares interfaces that game environments and applications can implement to plug into the minimap view.
- **FxMinimapView.h/.cpp** – Provides types used by the minimap view.

3.1.3 Game Simulation

- **HUDAdapter.h/.cpp** – Implementation of adapter design pattern to decouple the simulation from the HUD to allow for convenient reusability.
- **HUDEntityController.h/.cpp** – Declares controller types for entities, primarily used to update simulation logic and AI behavior.
- **HUDSimulation.h/.cpp** – Provides types which implement a demo environment to drive the game's UI. This environment contains players from two teams in a capture and hold game type.

3.2 Flash

The Flash content for the HUD Kit is divided in two files: **HUDKit.fla** and **Minimap.fla.** These FLA files define and layout the HUD Kit's UI elements for manipulation using Scaleform. They also include all images and icons displayed in the UI.

Both files are divided into several layers. Generally, each component or group of similar components has its own layer. Layers provide a convenient method for author-time control of element ordering based on depth. The top most layer will display on top of all other layers and so on.

In Scaleform, Flash animations can be handled using C++, ActionScript, or the Flash timeline. In this demo, the majority of HUD animations are handled on the Flash timeline via Classic Tweens. These animations are generally triggered by a GFx::Value::GotoAndPlay() call from C++.

3.2.1 HUDKit.fla

The HUDKit.fla file located in the *Bin\Data\AS2 or AS3\Kits\HUD* directory is the demo's primary SWF. This file is loaded by the HUDKitDemo application at runtime. Every Flash UI element exists in this file except for the minimap view which is loaded by the HUDKit.fla at runtime.

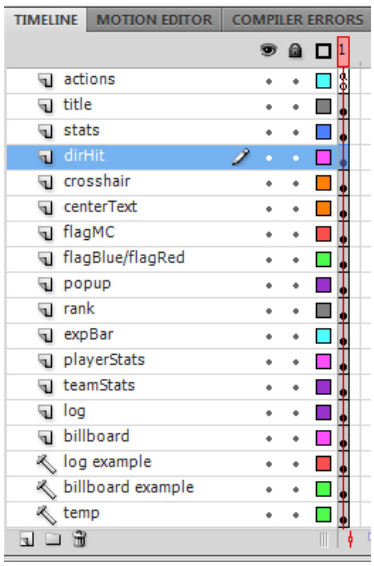

Figure 2: HUDKit.fla layers

The HUDKitDemo registers the HUDKit MovieClip using an External Interface call on Frame 1 of the timeline:

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

This passes a pointer to the MovieClip which C++ can register with the C++ HUDView to manipulate the user interface.

Every layer, its MovieClips, layouts, animations, and Scaleform C++ managers will be discussed in detail in the HUD View section of this document.

3.2.2 Minimap.fla

Figure 3: Minimap Symbol

Minimap.fla includes the 'Minimap' symbol, the core of the Minimap. For the scope of this demo, the background is a static image and not an interactive 3D environment.

The player is denoted as a yellow arrow in the center, and heading is shown by the minimap border (North is marked). In addition to the player, the minimap view displays five icon types:

- *Friendly players (blue)*. The directional arrows appear only at a high zoom level.
- *Enemy players (red)*. The directional arrows appear only at a high zoom level.
- *Capture points* (denoted by the flags: white for neutral, blue for friendly, and red for enemy)
- *Power-ups* (rotating green and white icon)

The capture points and powerups are sticky icons that stick to the view border when outside the view range but inside the detectable range. The Direct Access API method is used to provide the functionality to fade icons that enter and leave the detectable range. It is also used to attach and remove MovieClips from the Stage during runtime using C++.

This 'Minimap' symbol contains several layers. Layers provide a convenient method for author-time control of element ordering based on depth. The top most layer will display on top of all other layers and so on:

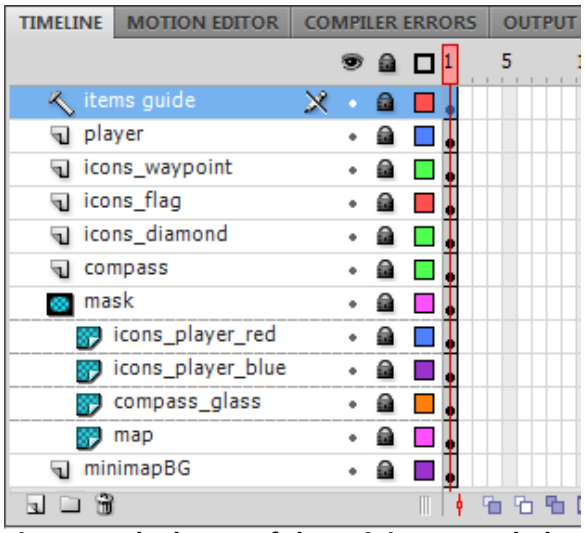

Figure 4: The layers of the 'minimap' symbol

- **items guide**: A guide layer useful for designers/artists. This layer's content is not published to the SWF file.
- **player**: The layer containing the yellow player icon.
- **icons_waypoint**: An empty canvas where waypoint icons are generated and cached.
- **icons_flag**: An empty canvas where flag icons are generated and cached.
- **icons_diamond**: An empty canvas where the power-ups' diamond icons are generated and cached.
- **compass**: Layer containing the compass (with North marked).
- **mask**: Mask layer that only shows the content inside the unmasked area.
    - **icons_player_red**: An empty canvas where red enemy icons are generated and cached.
    - **icons_player_blue**: An empty canvas where blue friendly icons are generated and cached.
    - **compass_glass**: An embellishment that provides a glassy sheen in the top left of the compass.
    - **map**: The layer containing the terrain map. For demo purposes we use a large bitmap for the terrain, however in production situations this will most probably be a texture that is updated from C++. In this case, the update logic does not need to change the map's offset/rotation, which is required when a masked terrain map is used.
- **minimapBG**: Contains a blue background that blends well with the terrain bitmap edges. This provides a smooth transition when the terrain bitmap goes out of view. This background is not required if the terrain map is rendered as a texture and updated directly from C++.

The 'Minimap' symbol uses the com.scaleform.Minimap class (located under *Bin\Data\AS2 or AS3\Kits\HUD\com\scaleform*). This minimal class defines the path to the map image and the constructor of the minimap which registers the minimap with the HUDKitDemo application and loads the map image.

4 HUD View

The C++ HUD View manages the content HUDKit.swf. Its interface declares the following types:

- FxHUDView – Core manager for the HUD (specifically, HUDKit.swf). Updates view and all child MovieClips based on the state of the simulation.
- BillboardCache – Manager and cache system for friendly/enemy billboards (center, onscreen indicators not to be confused with minimap icons).
- FxHUDLog – Manager for message log displayed at the left of the HUD. Tracks, displays, and reuses FxHUDMessages.
- FxHUDMessage – A message definition, which includes a MovieClip reference and message text.

Note that FxHUDView's cached MovieClip references are denoted by the *MC* suffix, short for MovieClip.

During its initialization, FxHUDView registers a reference to every major MovieClip that is changed in UpdateView() at runtime. This is essential for avoiding the retrieval of MovieClip references in every update, thus minimizing the time spent updating the view. The initialization also hides a number of unused UI elements defined in HUDKit.fla.

FxHUDView::UpdateView() updates every element of the view with the data provided by the simulation environment. Its logic is divided into the following sub-methods: UpdateTeamStats(), UpdatePlayerEXP(), Update PlayerStats(), UpdateEvents(), UpdateTutorial(), and UpdateBillboards(). Each of these methods is passed a pointer to the simulation environment which is then used to update the specific UI elements.

Note that MovieClip references are rarely retrieved during an UpdateView() call. Unnecessary MovieClip updates are avoided whenever possible, as updating MovieClips every frame is a potential waste of cycles.

With the Direct Access API, data of any type and form can now be passed efficiently between Flash and the application. The GFx::Value class in particular presents a number of new functions designed to give Scaleform users greater control over Flash content. GFx::Value::SetDisplayInfo(), GFx::Value::GetDisplayInfo() and GFx::Value::SetText() are used throughout the HUD View's update logic to directly and efficiently manipulate the display state of MovieClips.

4.1 Round Statistics and Scoreboard

The *teamStats* MovieClip, located in the teamStats layer in Scene 1 of HUDKit.fla, is the scoreboard and statistics for the round in progress. It includes the winning and losing textFields (*redWinning*, *blueWinning*),

the score for each team displayed in two textFields (*scoreRed*, *scoreBlue*) and progress bars (*teamRed, teamBlue)*, and the roundtime clock textField (*roundTime*).

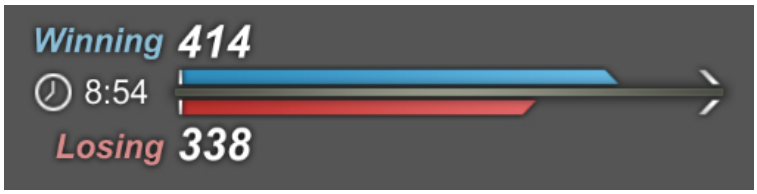

Figure 5: teamStats Symbol

teamStats is updated exclusively using FxHUDView::UpdateTeamStats(). The following code updates the *teamStats* MovieClip's elements.

```
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;
	}
}
```

The score is stored in a Scaleform::String and passed to the *scoreRed* and *scoreBlue* textFields using GFx::Value::SetText(). The X scale of the teamRed and teamBlue score bars are updated using GFx::Value::SetDisplayInfo(). Both begin at 0% of their original width and are scaled toward 100% as the teams' score increases toward the score needed to win the round.

Before updating the *blueWinning/redWinning* textFields, the two team scores are compared to avoid unnecessary updates. GFx::Value::SetText() is called to set the text using the constant "Winning" and "Losing" strings defined during the HUDView's initialization.

Before updating the round time textField, the time remaining at the last UpdateView() call is compared against the latest time. If more than a second has elapsed since the last update, the *roundTime* textField is updated with the formatted time String using GFx::Value::SetText().

4.2 Player Stats and Ammo

The *playerStats* symbol, located in the playerStats layer of Scene 1 of HUDKit.fla, contains the health textField and health bar shape (*healthN, health)*, weapon icon symbol (*weapon*), ammunition indicators for the HUD, and a low ammo indicator (*reload*).

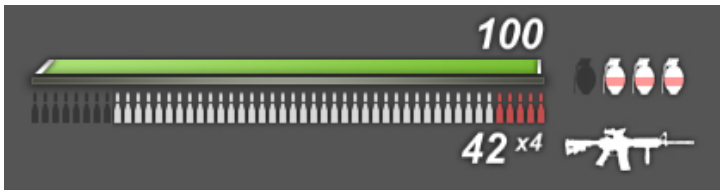

Figure 6: playerStats Symbol

The ammunition indicators are broken into six MovieClips, *clipN, ammoN, ammo, ammoBG, grenade,* and *grenadeBG.* textFields *clipN and ammoN* display the amount of remaining ammunition in the form of magazine clips for the weapon and loaded ammo.

The ammo/healthVehicle layer contains two symbols, *ammo* and *ammoBG,* each separated into an appropriately sub-layer. Both symbols contain ammunition indicator shapes below the health bar. Each of these symbols is divided into 51 Keyframes, each with one less bullet indicator than the previous Keyframe. GFx::Value::GotoAndStop() is used to visit different Keyframes of these MovieClips, changing the number of shapes displayed to reflect the state of the simulation.

- *ammo* displays the number of loaded bullets in the equipped weapon.
- *ammoBG* displays the maximum number of bullets in a clip for the equipped weapon.

The grenade/armor layer is set up similarly to the ammo layer with two symbols, *grenade and grenadeBG*, each separated into its own layer. Both symbols contain grenade indicator shapes. As in the ammunition indicators, each of these symbols is divided into 5 Keyframes. Frame 1 has 4 grenade shapes and Frame 5 has none. GFx::Value::GotoAndStop() is used to visit different Keyframes of these MovieClips, changing the number of shapes displayed to reflect the state of the simulation.

- *grenade* displays the number of grenades remaining.
- *grenadeBG* displays the maximum number of grenades the player can carry.

The following logic updates the *playerStats* symbol.

```
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 = UInt(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;
      }
}
```

In *ammo*, Frame 1 has 50 bullet indicators and Frame 51 has 0 bullet indicators. Each time the player fires his weapon, ammoMC.GotoAndStop(ammoFrames - ammoInClip) is called to visit the appropriate Keyframe, removing one loaded bullet indicator from the HUD.

For *ammoBG*, ammoBGMC.GotoAndStop() is used to visit the Keyframe that displays X number of bullet indicator backgrounds. Here, X is based on clip size of the equipped weapon. For example, the Machine Gun has a clip size of 50. Therefore ammoBG.GotoAndStop(1) will display all 50 bullet indicator backgrounds shown in Frame 1. This MovieClip is only updated when the equipped weapon has changed since the last FxHUDView::UpdateView() call.

4.3 Flag Capture Indicator

The *flagMC* symbol, located in the flag layer of Scene 1 of HUDKit.fla, is the container for the flag capture indicator. *flagMC* contains the *flag* symbol, which is divided into two parts, the flag capture indicator bitmap and the arrow indicator (*arrow)* which moves left and right based the CaptureState of the flag. The indicator fades-in when a player is near a capture objective and will fade-out when the player is no longer within range.

Figure 7: flagMC Symbol

The fade-in and fade-out animations for *flagMC* are done on the Flash timeline using a Classic Tween. *flagMC*'s timeline has 4 figurative states represented by 5 Keyframes: Hidden, Visible, Fade-In, and FadeOut. These states are visited using GotoAndStop() and the Keyframes' labels/numbers (1, 5, "On", "Off", etc…).

FxHUDView::UpdateEvents(), copied below, controls *flagMC*'s states and updates *arrow*'s location using the simulation's data.

```
// 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;
      }
}
```

When flagMC.GotoAndPlay("on") is called, *flagMC* plays its fade-in animation and then stop in the Visible state. When flagMC.GotoAndPlay("off") is called, *flagMC* plays frames 11-20, a fade-out animation. When the animation reaches Frame 20, it will automatically restart playback from Frame 1. This is default Flash behavior; no extra ActionScript or C++ is necessary. Because Frame 1 contains a `stop();` call, the MovieClip will stop after playing Frame 1. This is ideal, since the symbol is hidden in Frame 1 as its Alpha is set to 0.

The *arrow* is shifted horizontally using GFx::Value::SetDisplayInfo(). This method is passed a new GFx::Value::DisplayInfo with updated x-coordinate based on the CaptureState of the entity.

4.4 Weapon Reticule

The *reticule* symbol, located in the crosshair layer of Scene 1 of HUDKit.fla, is the container for the firing reticule. *reticule* contains four symbols, *left, right, bottom,* and *top*, each in its own layer. Having each piece of the reticule in its own symbol allows for individual manipulation. The reticule responds to PlayerFire events fired by the simulation whenever the user player fires his weapon. As the player fires a weapon, the reticule will expand dynamically. When the player stops firing, the reticule will return to its original position.

Figure 8: reticule symbol

The reticule update code is divided into two parts, FxHUDView::UpdateEvents() and FxHUDView::OnEvent() methods. The ReticuleHeat and the ReticuleFiringOffset are incremented and set, respectively, in FxHUDEvent when a PlayerFire Event is caught.

```
// 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() uses these two variables to shift the *top, bottom, left,* and *right* MovieClips. For each MovieClip, we use SetDisplayInfo() to shift the X or the Y coordinate of the MovieClip. The first number in the equation, 6 or -6, is the starting X or Y offset from the 0, 0 coordinate. Notice that single instance of GFx::Value::DisplayInfo is reused for each SetDisplayInfo() call. However, to ensure that no DisplayInfo is accidently reused, DisplayInfo.Clear() is called after each GFx::Value::SetDisplayInfo() call. This method clears all previously defined display properties for a DisplayInfo instance.

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

For this demo, a simple dynamic reticule equation was chosen for simplicity's sake, but this code can easily be modified to create more dynamic and creative reticule behavior.

4.5 Directional Hit Indicator

The *dirHit* symbol, located in the dirHit layer of Scene 1 of HUDKit.fla, is the container for the direction hit indicator. *dirHit* contains eight MovieClips, divided into appropriately named layers: *tl, l, bl, b, br, r, tr,* and *t*.

Each of these MovieClips contains an alpha-blended red semi-circle which will be used as an indicator for:

a) the user player taking damage
b) the relative direction of the source of that damage

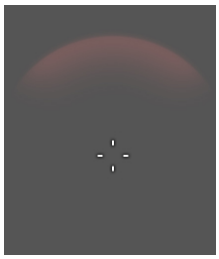

Figure 9: dirHit Symbol

When a player is hit, the hit indicator will appear and then fade-out after ~1 second. The fade-out animation is a Classic Tween alpha-blend on the Flash timeline. By calling GFx::Value::GotoAndPlay("on") for any of the hit indicator MovieClips, it will appear and then fade-out over 1 second. The appropriate MovieClip based on the relative direction of the damage's source.

```
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 Rank and Experience Display

The *expBar* symbol, located in the expBar layer of Scene 1 of HUDKit.fla, is the exp bar at the bottom of the screen which tracks the player's progress through his/her current rank. *expBar* contains the *exp* MovieClip which is the yellow bar that grows as the player gains experience through kills and captures.

The *rank* symbol, located in the rank layer of Scene 1 of HUDKit.fla, shows the player's current rank. *rank* contains 18 Keyframes, each dedicated to one rank. Each Keyframe displays the icon and title associated with the rank.

Figure 105: expBar and rank Symbols

```
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);
      }
}
```

Similar to the team score bars, *exp* is scaled based on the player's current experience when the player's experience changes. To update *rank*, GFx::Value::GotoAndStop() is used to Keyframe X, where X is the player's current rank + 1 (since there is no Frame 0).

4.7 Text Notifications

The *centerTextMC* symbol, located in the centerText layer of Scene 1 of HUDKit.fla, is an animated textField used to display information to the user in the center of the screen. In this demo, *centerText* is used to display a simple tutorial at the start of every round and to announce that the user player recently was killed. After the text is displayed it will fade-out.

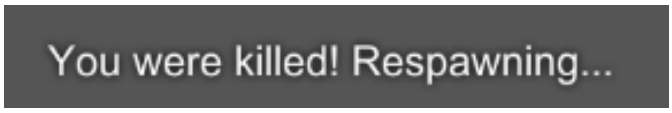

Figure 6: centerTextMC Symbol

centerTextMC contains the *centerText* MovieClip which contains the actual textField, *textField*. The MovieClips are arranged this way to allow for animation. *centerTextMC's* timeline contains two labeled Keyframes, "on5", which displays text for 5 seconds before playing a fade-out animation, and "on" which displays text for 3 seconds before playing a fade-out animation. The end of the timeline restarts the MovieClip's playback at Frame 1, where the *centerTextMC's* alpha is 0, hiding it until "on" or "on5" is called again.

```
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.
```

Here the *centerTextMC's textField* is set appropriately and the MovieClip is displayed using GotoAndPlay("on") The MovieClip will fade-out on its own after ~3 seconds because of the timeline's Classic Tween.

4.8 Pop-ups and Notifcations

The *popup* symbol, located in the popup layer of Scene 1 of HUDKit.fla, is an animated event indicator. In this demo, *popup* is played whenever the user player reaches 3 or more consecutive kills (a kill streak). It will fade-in and play an animation when displayed and it will fade out after 3-5 seconds based on the C++ call.

Figure 7: popup Symbol

Popup is the core container for the pop-up. All of the animation for its sub-elements is done on *popup's* timeline using Classic Tweens. *popup* contains 4 layers, *actions, popupNumber, popupText* and *popupBG*. *popupBG* is the animated black shape that "explodes" behind the number. *popupText* and popupNumber contain the *textField* symbols that display the text and number for the pop-up message. This symbol hierarchy is necessary to animate the textFields on the timeline.

The symbol has a "on" Keyframe, which can be called using GotoAndPlay() to start the animation. Once the initial animation is complete, the pop-up will fade out and playback will return to Frame 1 where it will stop, hidden with its Alpha set to 0.

```
// 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;
```

Here, the FxHUDEvent is cast as a KillStreak event. The number of consecutive kills is then retrieved from the player and formatted appropriately. The text for the pre-cached *popupText*'s *textField* symbol, PTextFieldMC, is set via C++ and GotoAndPlay("on") is used to trigger *popup*'s display animation.

4.9 Message and Event Log

The *log* symbol, located in the log layer of Scene 1 of the HUDKit.fla, is the container for the message and event log at the bottom-left side of the HUD. In this demo, the log displays text messages related to recent events including kills, power-ups, flag captures, and level-ups. New messages are added at the bottom of the log, forcing older messages upward. A log message will fade-out after ~3 seconds of being displayed.

Figure 138: log and logMessage symbols

Log messages are instances of the *logMessage* symbol. Messages added to the log will display for approximately 3 seconds before beginning a fade-out animation. This animation is defined on the timeline of the *logMessage* symbol. The text for the messages is sent from C++ and is displayed in the textField using htmlText.

The following code is from `FxHUDMessage::Movieclip* FxHUDLog::GetUnusedMessageMovieclip()` which creates a new LogMessage MovieClip and attaches it to the Log MovieClip, the canvas for all log messages.

```
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;
}
```

The content for the message is set from C++ based on the event type. The following is the C++ code to process a Level Up event and set the htmlText for a new message that will display "You are now a [Rank Icon] [Rank Name]" in the log.

```
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++;
```

The following is the HUD Update logic used to manage the log. It updates the layout and animation for log message references.

```
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;
      }
}
```

The update logic for the Log checks to see whether any FxHUDMessage is currently displayed in the log by checking whether its Alpha is set to 0. If it is not displayed, it is removed from the Log and added back to the MessageMCs list of unused FxHUDMessage MovieClips. If a new FxHUDMessage has been added, all MessageMC are shifted up appropriately using GFx::Value::DisplayInfo.SetY() and GFx::Value::SetDisplayInfo().

4.10 Billboards

The *billboard_container*, located in the billboard layer of Scene 1 of HUDKit.fla, is the container for player billboards. Each billboard displays an arrow and player name that follows the player's location. Billboards are often used in games to show additional detail of 3D objects that are contained within the view frustum of the user player. In this demo, player billboards appear at a certain distance from the user player. Friendly player billboards always display player names while enemy billboards only display the arrow until the target is close to the view center.

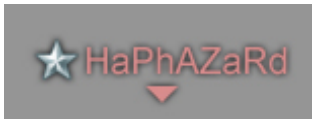

Figure 9: billboard_enemy Symbol

The billboard implementation in this demo creates a 2D billboard MovieClip in the UI which is shifted and scaled to follow players' positions using GFx::Value::SetDisplayInfo().

Each billboard MovieClip (*billboard_friendly, billboard_enemy)* is constructed from one *billboard_label_(friendly/enemy),* located in text layer, and one arrow image, located in the arrow layer. *billboard_label_** contains a textField which is used to display the target player's name. *billboard_friendly'*s "Show" Keyframe starts a fade-in animation using a Classic Tween alpha blend on *billboard_friendly*'s timeline. Once "Show" is played, the billboard will remain visible until it is hidden or removed.

The C++ logic to attach billboard MovieClips to the canvas is in `FxHUDView::BillboardCache::GetUnusedBillboardMovieclip`:

```
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);

}
```

Each BillboardCache contains is a reference to the MovieClip to which it attaches billboards named CanvasMC. This reference is passed to the BillboardCache when it is instantiated. The logic above attaches a billboard MovieClip of the symbol billboard_enemy/billboard_friendly to its predefined canvas MovieClip at the CanvasMC.AttachMovie() call and retains a reference to it for reuse.

The Billboard update logic is defined in the following methods: UpdateBillboards(), BeginProcessing(), EndProcessing() and GetUnusedBillboardMovieclip(). BeginProcessing() and EndProcessing() are called at the beginning and end of updating a set of friendly or enemy billboards. These methods manage the lists of used and unused billboards.

The following code is the update logic for friendly billboards from UpdateBillboards().

```
// 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() creates and updates billboards based on a list of entities retrieved from a user player view query. It uses GotoAndPlay("show") to display billboards, GFx::Value::SetDisplayInfo() to set the MovieClip's X, Y, and scale, and GFx::Value::SetText() to change the text displayed.

The textField uses htmlText to populate and display the billboard. The text field contains an image of the player's rank and the player's name. The rank icon is defined by the *src='rank{0}'* HTML. In the example above, it corresponds to the friendly player's rank + 1 (`friendlies[i].pPlayer->GetRank()+1`). The other *img* options define the icon image's height, width, alignment, and vspace. The player's name is defined by the {1}. In the example above, it corresponds to the name of friendly player (`friendlies[i].pPlayer->GetName()`).

5 Minimap View

The C++ side of the Minimap consists of the following files (located in *Apps\Kits\HUD*\):

- **FxMinimap.h**: Declares interfaces that game environments and applications can implement to plug into the minimap demo core.
- **FxMinimapView.h/.cpp**: Provides types used by the minimap view.

5.1.1 The Minimap View

The minimap view interface declares the following types:

- **FxMinimapIconType**: An icon type definition, which includes a type ID and an extra sub-type value for additional specialization.
- **FxMinimapIcon**: Icon resource that is linked to an element in the view.
- **FxMinimapIconCache**: Icon cache that stores sets of icons of a specific type. The icon view states are managed by the cache as needed, and the cache also supports fading in/out of icons that enter/exit the detectable range. The cache uses a factory to generate new icons. The base implementation of the factory is a templatized class called MinimapIconFactoryBase. Each factory handles attaching the icons' MovieClips to its appropriate canvas MovieClip as necessary and removing these MovieClips when the Minimap is destroyed. Multiple factory types are defined to support the different icon types listed below. Each icon type contains an Update() method that implements icon type specific update logic, such as translation, rotation and textField changes:
    - *PlayerIcon*: Represents both friendly and enemy icon resources because both icon types share the same logic. They only differ by the icon creation methods.
    - *FlagIcon*
    - *ObjectiveIcon*
    - *WaypointIcon*
- **FxMinimapView**: The main controller for the minimap view. Its UpdateView() method is called by the application to refresh the view.

The following code is the PlayerIcon::Update() method (FxMinimapView.cpp; line 116). It shows the logic used to update the player icons in the view using the Direct Access API. The *MovieClip* member contains a reference to the icon MovieClip on stage. The SetDisplayInfo method directly modifies the MovieClip's display properties (the display matrix, visibility flag or alpha value) instead of going through the slower SetMember() method. Note that GFx::Value::SetMember is still faster than GFx::Movie::SetVariable:

```
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 Performance Analysis

The statistics displayed by the application represent the time spent updating the HUD view, the time spent updating the Minimap, the number of minimap objects updated, the total display time for the Scaleform content and the time spent advancing the SWF.

The number of objects is the accumulation of all types of entities including friendly/enemy players, power-ups, and objectives. The update time for the HUD includes updating all the UI indicators such as the billboards, log messages, round scoreboard, and the player's stats. The update time for the Minimap includes the time spent updating the aforementioned minimap objects, as well as the masked terrain, the compass and the player icon. Updating these views consists of executing all logic governing the visual properties of the MovieClips, such as x/y position, label, current frame, rotation, scale, and transform matrix.

6.1 Performance Statistics

Previous Scaleform implementations were limited to using the C++ GFx::Movie::SetVariable and GFx::Movie::Invoke methods to update a Movie's view state. This unfortunately incurred a significant performance penalty due to many factors, such as resolving MovieClip paths. In contrast, the Direct Access API provides faster code paths to perform the same functionality, thus making complex HUD updating much more efficient using Scaleform 3.1 and later.

The following graph shows the significant increase in the Minimap's performance (lowering of the update time) using the Direct Access API method compared to the SetVariable/Invoke method. The performance gain is in the order of 10-25x:

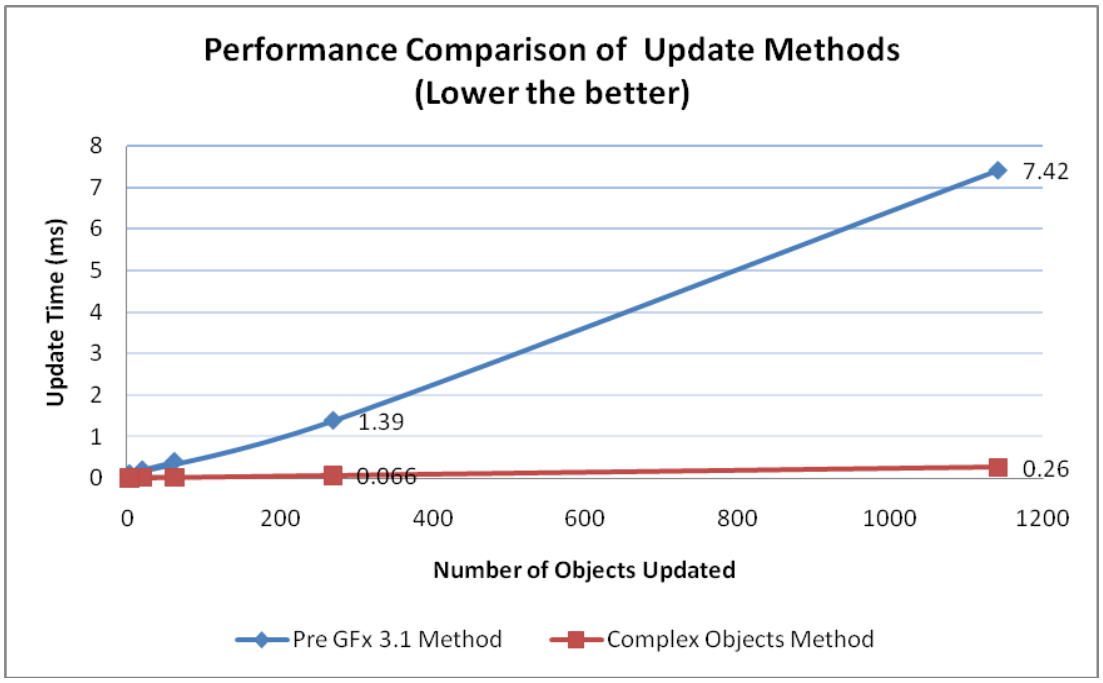

Figure 10: Performance comparison of the Direct Access API vs. SetVariable/Invoke update methods for Minimap

As shown above, the update time of the Direct Access method is optimal for all working environments as it is well below the 1ms threshold – the average allocated processing time for HUD UIs.

Average performance statistics for the whole HUD Kit over the course of one 15-minute simulation round are presented below by platform. These numbers provide an estimate for performance of a fully-featured, Flash HUD driven by Scaleform:

Platform	FPS	Update (ms)	Display (ms)	Advance (ms)	Total (ms)
Windows Vista on a MacBook Pro	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

Table 1: Per Platform Average Performance Statistics for Scaleform HUD Kit

As shown in Table 1, the update time for HUD is optimal for all working environments as it is well below the 1ms threshold – the average allocated processing time for HUD UIs.

6.2 Memory Breakdown

Below is a memory breakdown for all uncompressed content loaded by the HUD Kit demo. Content was exported using GFxExport's default settings. The total memory foot footprint includes .GFX files, images, components and exported fonts.

1. **File Format: GFX Standard Export, Images Uncompressed**

Total Memory: 1858kb

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

Uncompressed Images

- HUDKit.gfx
  - Core UI Components - 603kb
  - Rank Icons - 72kb
  - Weapon Icons - 51kb
    - **Note:** 43kb of the Weapon Icons are unused in this version of the HUD Kit.
- Minimap.gfx
  - Components - 644kb
- Map.jpg - 259kb