Scaleform GFx

# Scale9Grid User Guide

This document describes the details of using Scale9Grid functionality in Scaleform GFx to create resizable windows, panels and buttons.

Authors: Maxim Shemanarev, Michael Antonov

Version: 1.02
Last Edited: November 2, 2010

Scaleform®

## Copyright Notice

How to Contact Autodesk Scaleform:

| | |
|---|---|
| Document | Scale9Grid User Guide |
| Address | Scaleform Corporation<br>6305 Ivy Lane, Suite 310<br>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: Scale9Grid in Flash and GFx

Scale9Grid is used in Flash® to specify component-style scaling for movie clips. Scale9Grid allows developers to create movie clip symbols and user interface components that resize intelligently to make use of the provided screen real estate, instead of scaling linearly which is commonly done with graphics and design elements.

In Flash terminology, Scale9Grid is also known as 9-slice scaling. When 9-slice scaling is enabled, the movie clip is conceptually divided into nine sections with a grid-like overlay, such that each one of the nine areas is scaled independently. To maintain the visual integrity of the movie clip, corners are not scaled, while the remaining areas of the image are scaled (as opposed to being stretched).

In the Flash Studio, 9-slice scaling is available as a checkbox in the movie clip “Symbol properties” dialog box. In addition, MovieClip.scale9Grid and Button.scale9Grid properties are also exposed in ActionScript to give programmatic control over piecewise scaling. For more information on the use of Scale9Grid in Flash, please refer to the Flash Studio documentation.

The Adobe Flash player has certain restrictions on Scale9Grid that make it relatively difficult to use in many practical situations. For example, the standard Flash player does not support images slicing and fails to process Scale9Grid with free rotation or other arbitrary transformations. Scaleform GFx supports Scale9Grid in a more consistent manner, while keeping good compatibility with Flash. The Scale9Grid features of both Adobe Flash and Scaleform GFx are briefly described in the following table.

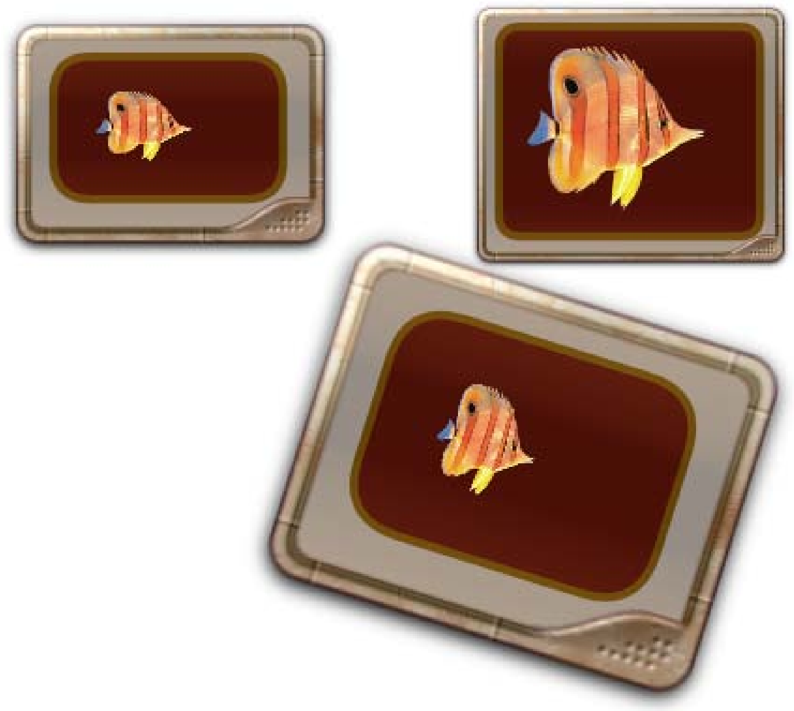
| **Scale9Grid Features** | **Adobe Flash Player** | **Scaleform GFx** |
|---|---|---|
| Transformations | X/Y Scale only | Arbitrary transformations |
| Bitmaps | Average scaling based on the bounding box | Automatic tiling according to the Scale9Grid |
| Gradients and image fill | Average scaling based on the bounding box | Average scaling based on the bounding box |
| Sub-symbols (nested movie clips, buttons, graphics) | Not supported | Supported |
| Nested Scale9Grids | Not supported | Not supported |
| Transformations of the enclosing symbol | Supported | Supported |
| Text | Scaled as usual | Scaled as usual |

Scaleform GFx preserves the backward compatibility with Flash, which means that the Scale9Grid functionality that works in Flash will behave the same way in GFx. However, the features that are not supported in Flash may behave differently, but in a more practical manner. For example, support for sub-symbols (nested movie clips) is vitally important for creating interactively resizable windows. The differences between the two player implementations are covered in detail throughout this document.

To provide developers with a hands-on example of Scale9Grid, this document refers to a number of sample FLA/SWF files.  These files are available from the Downloads section of Scaleform Developer Center website as  gfx_2.1_scale9grid_sample.zip, which can be used for all GFx versions.

## 2 Transformations

Unlike Flash, GFx supports arbitrary transformations of the movie clip with applied Scale9Grid. Whereas, in Flash the referenced movie clip cannot be rotated or skewed.  However, it is possible to include the resulting movie clip in a sprite and then use a rotation or skew that enclosing sprite. Transformations of the enclosing sprite works identically in both, Flash and GFx.  Transforming the higher level symbols makes it possible to draw different looking shapes with the same Scale9Grid, as in the following example.



## 3 Scale9Grid in 3D Movie Clips

Scale9Grid, due to its nature is not quite compatible with 3D tiling and rotation, because the idea of keeping a permanent scale in the corners and sides contradict perspective transformations. However, in GFx it's possible to have correct scale9grid with 3D in the following way:

1. Create a Scale9Grid movie clip which must not have any 3D transformations, only 2D.
2. Next, create an extra, enclosing movie clip to which 3D transformations are applied.

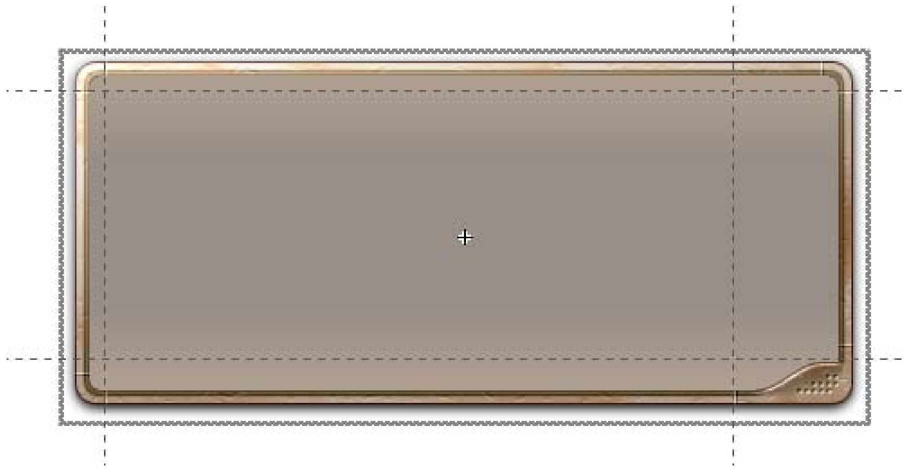
In other words, do not apply Scale9Grid and 3D directly to the same movie clip. The correct tree looks like this:  Root -> Movie Clip with 3D applied -> Movie Clip with shapes and Scale9Grid applied.

# 4 Processing Bitmaps

Although gradient and bitmap shape fill styles are transformed identically in Flash and GFx, separate standalone bitmaps are processed differently.  In Flash, when a standalone bitmap is placed in a 9-slice scaled movie clip, the player ignores any of the Scale9Grid settings unless the bitmap is explicitly cut into nine pieces by the user.  Whereas, the Scaleform GFx player automatically slices the bitmap so it is scaled correctly with Scale9Grid.  This behavior greatly simplifies development and allows creation of resizable buttons and windows without manual image cutting by the artist.  Furthermore, automatic slicing eliminates the anti-aliasing seams that are possible when stitching content rendered with EdgeAA.  The behavior of automatic slicing is explained in more detail in the following example.

Suppose the artist creates a bitmap that represents a window background similar to the image below:

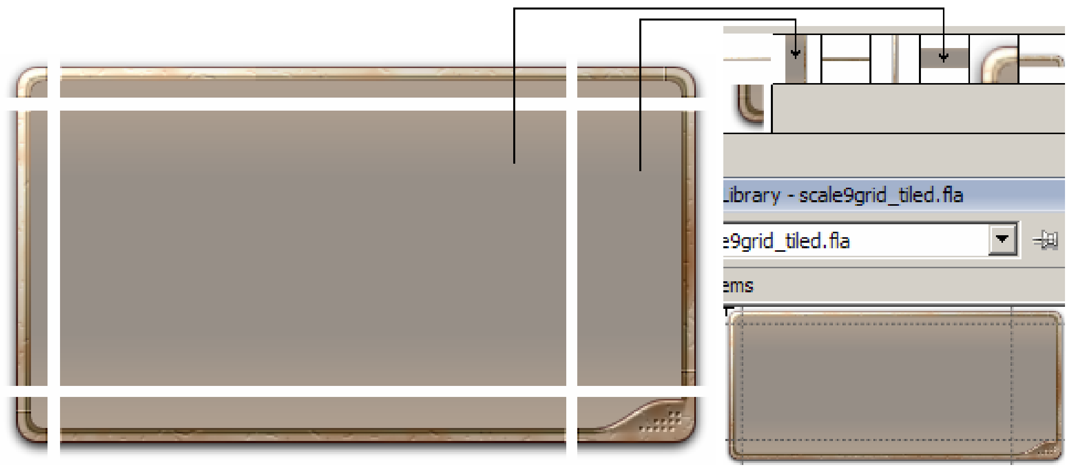
Then the artist applies Scale9Grid so that the shape of the corners can be preserved when scaling. The most obvious way to set this up is to create a movie clip and apply Scale9Grid to it as follows:



However, it will not work in Flash as expected. If you scale the resulting movie clip, it will look as if no Scale9Grid was applied:

In GFx this will work as expected, preserving the shapes of the corners:

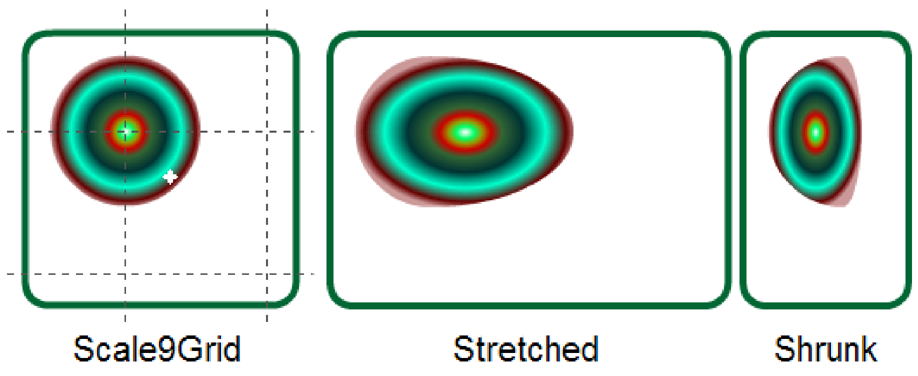
Adobe Flash calculates the average scale for the image on the basis of its bounding box. This means that it is still possible to make 9-slice scaling work with bitmaps, but it requires nine separate image pieces that correspond exactly with the Scale9Grid tiles:



Scaleform GFx does this operation automatically when playing the Flash file.

However, if the image is arbitrarily rotated or skewed, GFx will process it identically to Flash. An example of such “manually tiled” window is illustrated by files scale9grid_tiled.fla and scale9grid_tiled.swf.

Shapes with bitmap and gradient fill styles are displayed identically in Flash and GFx. Note that both gradient and bitmap fills fail to preserve the Scale9Grid transformation.  In effect, the transformation is only applied to the vector shape outline, but not to the fill style content. For example:



## 4.1 Previewing in Flash IDE

To automate the process of preparing bitmaps for 9-slice scaling, the *Cut Bitmap into Scale9 Slices* JSFL command is now included in the Scaleform Flash Extensions package for GFx 3.3.  Prior to this command, the only way to preview how a bitmap would scale, according to an applied Scale9Grid, would be to run the SWF file in GFxPlayer.  While the Scaleform Launcher now makes it relatively easy to view a SWF in GFxPlayer directly from Flash, not being able to view the bitmap properly scaled in the Flash IDE can be inconvenient to most artists.

In the screenshots below, a Scale9Grid is applied to a bitmap and then scaled in the Flash IDE.  The result is similar to what is produced when there is no Scale9 applied. To solve this issue, we take advantage of the available functionality provided by the Flash IDE to correctly scale vector shapes – in our case, bitmap fills. The command creates 9 separate shapes with the appropriate bitmap fills to match the source bitmap.  Once the command is applied, the Flash IDE is able to correctly preview scale9 shapes, and therefore is able to display the expected output as shown below.

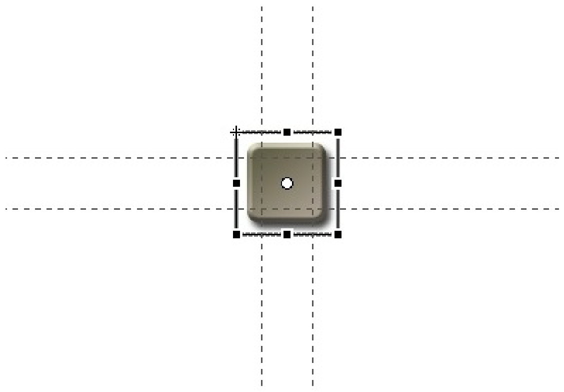


**Figure 1 – Adjusting the Scale9 grid**

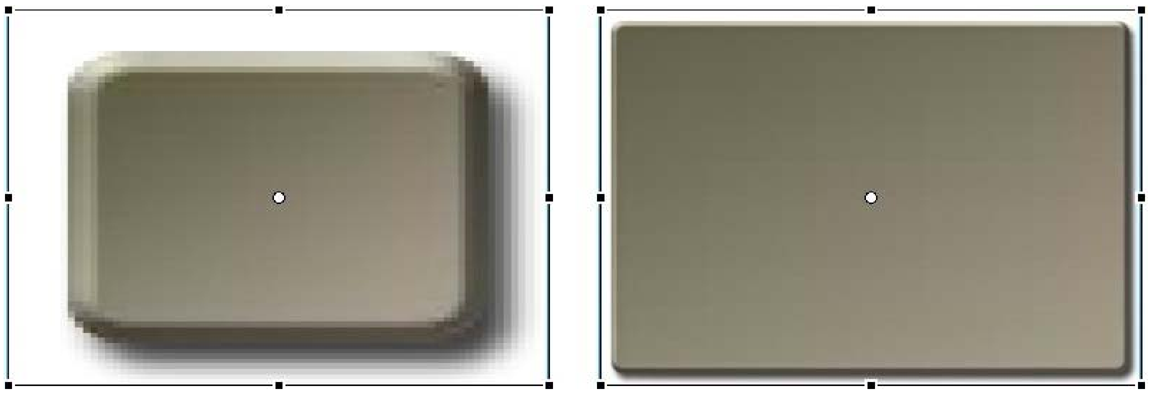


**Figure 2 – Before and after running the command**

### 4.1.1 Installation

The *Cut Bitmap into Scale9 Slices* command is installed by running the *Scaleform Extensions.mxp* file. Please see Getting Started with GFx for more information on how to install the Scaleform Extensions. The command will be available under the Flash IDE's *Commands* menu after installation.

### 4.1.2 Usage

The following steps demonstrate the proper use and functionality of the *Cut Bitmap into Scale9 Slices* command. It covers the necessary steps to prepare a bitmap for Scale9 slicing, as well as the behind-the-scene actions used to modify the bitmap to produce the correct output.

Create a new Flash project and then import a Scale9 applicable bitmap (as used in this example) onto the stage. Once the bitmap has been imported to the stage, convert the bitmap into a symbol. While in the "Convert to Symbol" screen, make sure to select "Enable guides for 9-slice scaling".

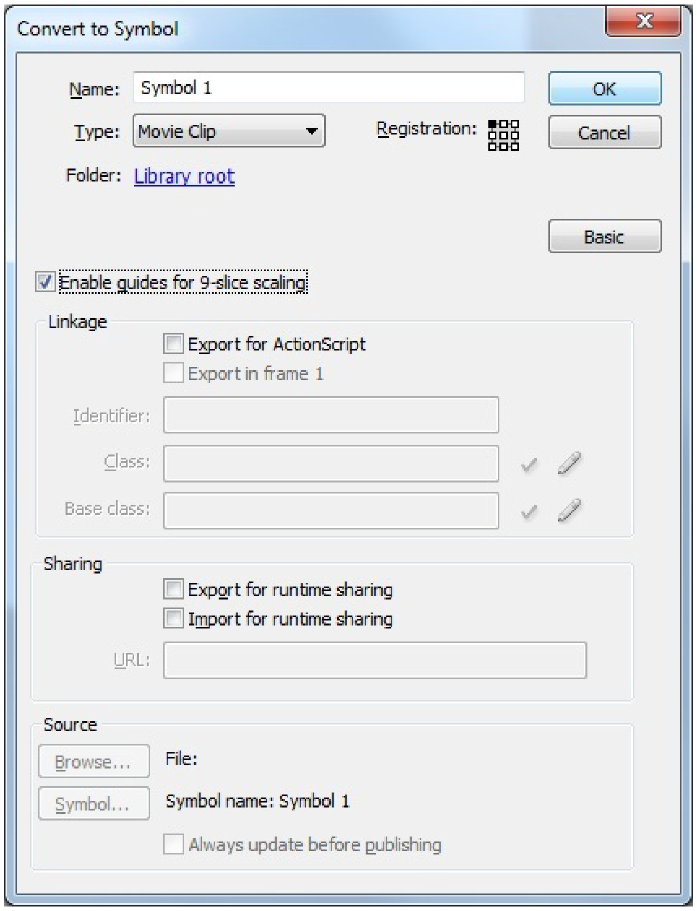


**Figure 3 - Enable guides for 9-slice scaling**

Once back on the main stage, select the "Free Transform Tool" and then scale the symbol instance. As seen in Figure 2, the bitmap does not scale correctly. If necessary, enter the symbol edit mode and adjust the Scale9Grid. After setting the desired Scale9 positions, select "Commands" from the main menu bar, and then select "SF - Cut Bitmap into Scale9 Slices."

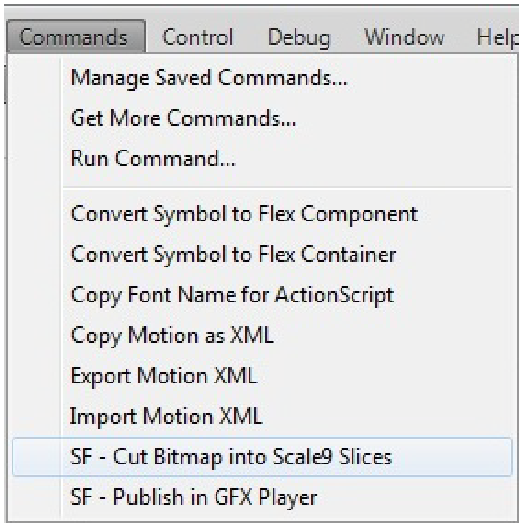

Commands
Control
Debug
Window
Help
Manage Saved Commands...
Get More Commands...
Run Command...
Convert Symbol to Flex Component
Convert Symbol to Flex Container
Copy Font Name for ActionScript
Copy Motion as XML
Export Motion XML
Import Motion XML
SF - Cut Bitmap into Scale9 Slices
SF - Publish in GFX Player

**Figure 4 – Location of the command**

The bitmap is copied to a new layer and automatically sliced into nine rectangles, each with their own bitmap fill, according the Scale9Grid.

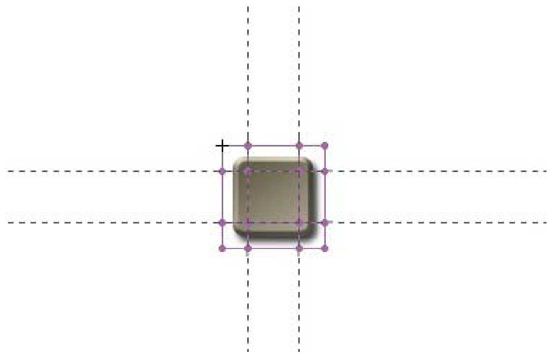


**Figure 5 - Command applied**

The command saves the original bitmap to a hidden guide layer. This enables the command to be run again if the Scale9Grid needs to be adjusted.

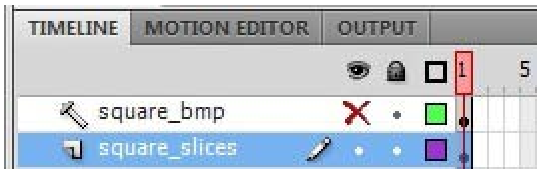


**Figure 6 - Layers created by the command**

Returning to the stage with the main Movie Clip, the bitmap is now properly scaled.  Please note that *Enable Live Preview* must be selected to view the bitmap with Scale9Grid applied in the Flash IDE. However, *Enable Live Preview* is not required to view the output in Flash Player or GFxPlayer.

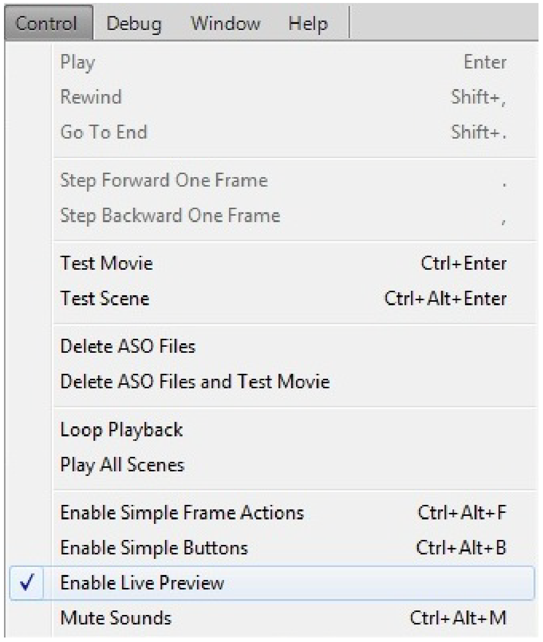


**Figure 7 - Enable Live Preview**

### 4.1.3 Known Issues/Important Notes

It is important to note that the *Cut Bitmap into Scale9 Slices* command only works in Flash CS4 and CS5. Earlier versions of the Flash IDE do not support the essential JSFL hooks to support the command.

Designers should not manually tweak the slices created by the command. Interacting with the shapes may cause the IDE to produce erroneous output that may be visible in both Flash Player and GFxPlayer. Selecting the generated shapes is not recommended.

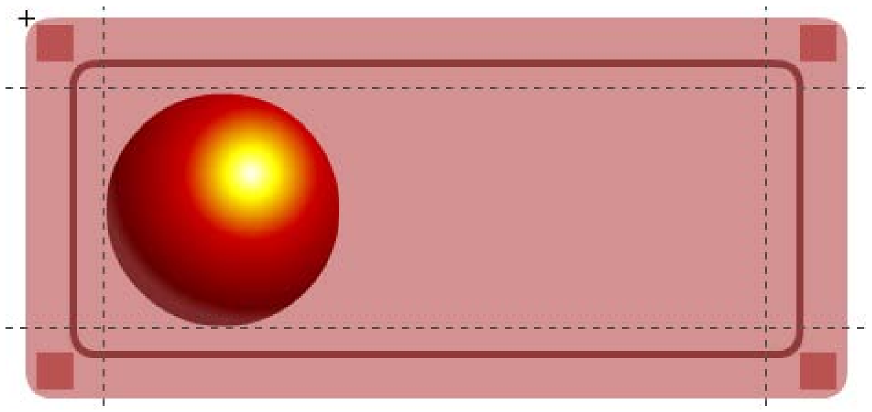
As noted in the section 3.1.2, *Enable Live Preview* must be selected to correctly preview the scaled bitmap. However, when using many symbols containing scale9Grids, some components may randomly become invisible in the Flash IDE. This is a known issue with the Adobe Flash IDE and is currently being investigated. The only solution at the moment is to turn off *Enable Live Preview*.

# 5 Interactive Windows

This section provides examples of interactively resizable windows and describes the ActionScript source code required to make them work with user input.

## *5.1 Flash Compatible Example*

The following example demonstrates how to create a Flash compatible version of an interactively resizable window. The example relies on the scale9grid_window1.fla and scale9grid_window1.swf sample files. Suppose we created the following movie clip in the Flash Studio.



The idea is to resize the window by dragging the little squares in the corners and move it by dragging any other part of its surface. The major problem here is that Flash and its ActionScript do not provide any mechanism for hit-testing that would distinguish between the different shapes and layers inside the movie clip. Furthermore, the 9-slice scaling does not work for nested movie clips in Flash. This contradiction makes it hard to program the resizing logic. As a work-around for hit-testing shapes, programmers have to explicitly check the coordinates. The example below illustrates the ActionScript program associated with the given movie clip.

```
import flash.geom.Rectangle;

var bounds:Object = this.getBounds(this);
for (var i in bounds) trace(i+" --> "+bounds[i]);
this.XMin = bounds.xMin;
this.YMin = bounds.yMin;
this.XMax = bounds.xMax;
this.YMax = bounds.yMax;
this.MinW = 120;
this.MinH = 100;
this.OldX = this._x;
this.OldY = this._y;
this.OldW = this._width;
this.OldH = this._height;
this.OldMouseX = 0;
this.OldMouseY = 0;
this.XMode = 0;
this.YMode = 0;


this.onPress = function()
{
      this.OldX = this._x;
      this.OldY = this._y;
      this.OldW = this._width;
      this.OldH = this._height;
      this.OldMouseX = _root._xmouse;
      this.OldMouseY = _root._ymouse;
      this.XMode = 0;
      this.YMode = 0;

      var kx = (this.XMax - this.XMin) / this._width;
      var ky = (this.YMax - this.YMin) / this._height;
      var x1 = this.XMin + 6  * kx;   // Left
      var y1 = this.YMin + 4  * ky;   // Top
      var x2 = this.XMax - 26 * kx;   // Right
      var y2 = this.YMax - 25 * ky;   // Bottom
      var w  = 20 * kx;
      var h  = 20 * ky;
      var xms = this._xmouse;
      var yms = this._ymouse;

      if (xms >= x1 && xms <= x1+w)
      {
            if (yms >= y1 && yms <= y1+h)
            {
                  this.XMode = -1;
                  this.YMode = -1;
            }
            else
            if (yms >= y2 && yms <= y2+h)
            {
                  this.XMode = -1;
                  this.YMode =  1;
            }
      }
      else
      if (xms >= x2 && xms <= x2+w)
      {
            if (yms >= y1 && yms <= y1+h)
            {
                  this.XMode =  1;
                  this.YMode = -1;
            }
            else
            if (yms >= y2 && yms <= y2+h)
            {
                  this.XMode =  1;
                  this.YMode =  1;
            }
      }

      if (XMode == 0 && YMode == 0)
      {
            this.startDrag();
      }
}


this.onRelease = function()
{
      this.XMode = 0;
      this.YMode = 0;
      this.stopDrag();
}

this.onReleaseOutside = function()
{
      this.XMode = 0;
      this.YMode = 0;
      this.stopDrag();
}

this.onMouseMove = function()
{
      var dx = _root._xmouse - OldMouseX;
      var dy = _root._ymouse - OldMouseY;

      if (this.XMode == -1)
      {
            this._x     = this.OldX + dx;
            this._width = this.OldW - dx;
            if (this._width < this.MinW || _root._xmouse > this.OldX + this.OldW)
            {
                  this._x     = this.OldX + this.OldW - this.MinW;
                  this._width = this.MinW;
            }
      }
      if (this.XMode == 1)
      {
            this._width = this.OldW + dx;
            if (this._width < this.MinW || _root._xmouse < this.OldX)
                this._width = this.MinW;
      }
      if (this.YMode == -1)
      {
            this._y      = this.OldY + dy;
            this._height = this.OldH - dy;
            if (this._height < this.MinH || _root._ymouse > this.OldY + this.OldH)
            {
                  this._y      = this.OldY + this.OldH - this.MinH;
                  this._height = this.MinH;
            }
      }
      if (this.YMode == 1)
      {
            this._height = this.OldH + dy;
            if (this._height < this.MinH || _root._ymouse < this.OldY)
                this._height = this.MinH;
      }
}
```

As you can see, the logic is rather complex. We need to have the initial bounding box of the movie clip and other variables. The important values that control movement are `XMode` and `YMode`. The mode value of '`-1`' means we are dragging the left or the top window border, respectively. The value of '`1`' means we are dragging the right or the bottom one.

The main hit-test logic is contained in the `onPress()` function. First, we have to calculate the scaling coefficients `kx` and `ky`, because the mouse coordinates do not automatically preserve the Scale9Grid transformation logic:

```
      var kx = (this.XMax - this.XMin) / this._width;
      var ky = (this.YMax - this.YMin) / this._height;
```

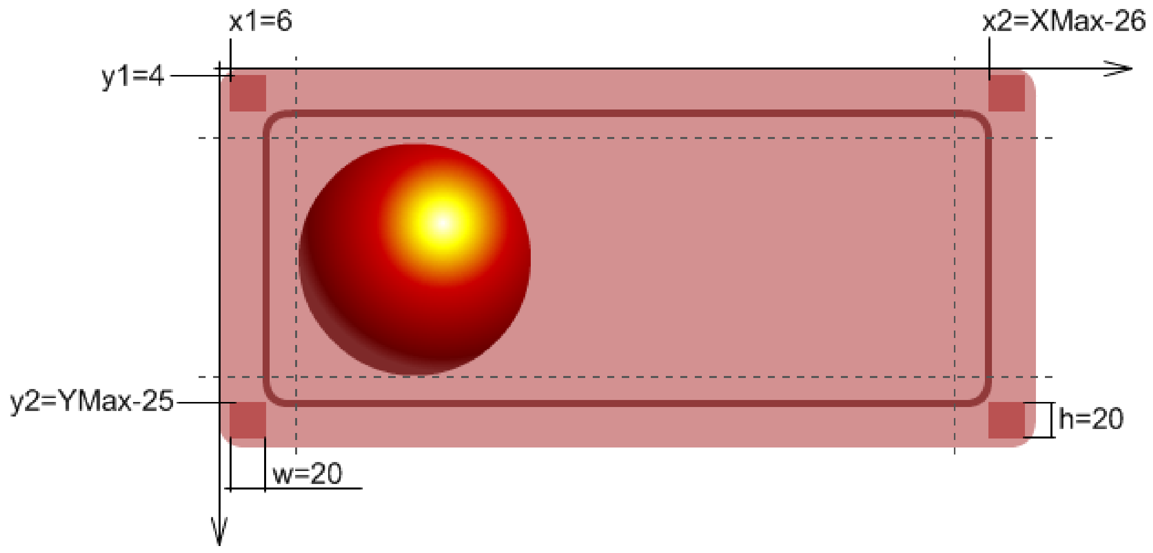
Then we calculate the hit-test rectangles (actually, squares in our case) which are:

| | |
|---|---|
| Left-Top: | x1, y1, x1+w, y1+h |
| Right-Top: | x2, y1, x2+w, y1+h |
| Left-Bottom: | x1, y2, x1+w, y2+h |
| Right-Bottom: | x2, y2, x2+w, y2+h |

```
      var x1 = this.XMin + 6  * kx;   // Left
      var y1 = this.YMin + 4  * ky;   // Top
      var x2 = this.XMax - 26 * kx;   // Right
```

```
    var y2 = this.YMax - 25 * ky;    // Bottom
    var w  = 20 * kx;
    var h  = 20 * ky;
```

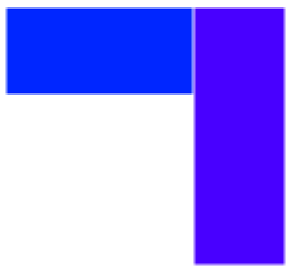
Note the constant values 6, 4, 26, 25, and 20. They are actually the coordinates that must exactly correspond with the respective shapes in the movie clip. In fact, the squares in the corners are not necessary and can be removed. This is the major disadvantage of the method: when changing the shapes you have to modify the ActionScript code accordingly. The following picture explains the meaning of the used constants.



In the next step we programmatically verify the coordinates and assign the respective resizing modes.

```
    var xms = this._xmouse;
    var yms = this._ymouse;
    if (xms >= x1 && xms <= x1+w)
    {
          ... and so on
```

Needless to say, the more complex shapes in the corners would require more complex logic. For example, the following corners would need to check for 2 rectangles each:



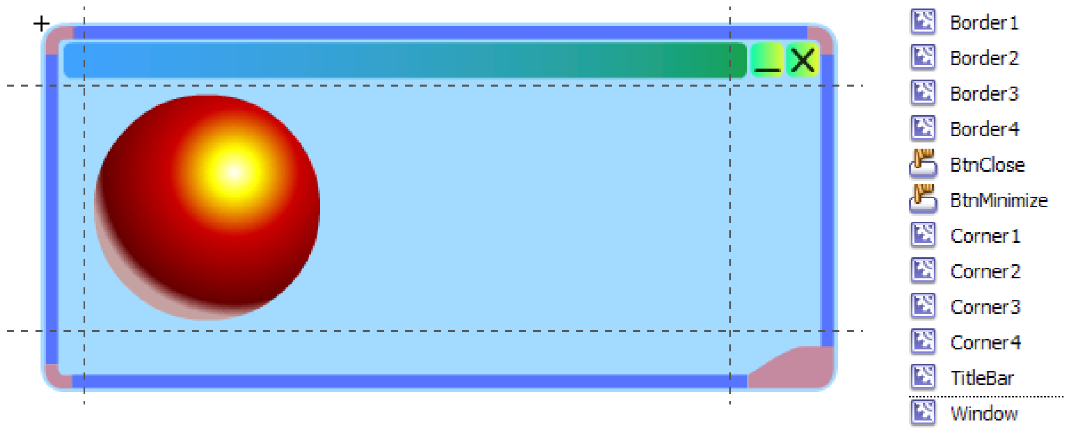
Checking for rounded corners or any other irregular shapes becomes very complex. Furthermore, this example does not handle resizing borders.

The rest of the ActionScript logic deals with changing the window coordinates and size while respecting the minimal width and height restrictions.

Note that a window with an image background requires manual tiling to be fully compatible with Flash.

## *5.2 Advanced GFx Example*

An example that works in GFx, but not in Flash is more advanced.  It relies on the fact that GFx transforms nested movie clips correctly, which means you can use the mouse hit-test functionality for arbitrary, irregular shapes.  It is included in files scale9grid_window2.fla and scale9grid_window2.swf. The graphics for this example are more complex, providing advanced functionality based on the generalized ActionScript code that does not use hard-coded coordinates and can be easily reused.



The sample Flash file consists of separate movie clips and buttons, such as Border1 to Border4 (blue), Corner1 to Corner4 (pink), TitleBar, and so on.  Since Scaleform GFx adjusts the nested movie clips correctly based on Scale9Grid, all you have to do is assign functions to the movie clips. The ActionScript becomes simple and obvious.  Note that the `OnResize()` function is the same as in the previous example.

```
import flash.geom.Rectangle;
//trace(this.scale9Grid);
this.MinW = 100;
this.MinH = 100;
this.XMode = 0;
this.YMode = 0;
this.OldX = this._x;
this.OldY = this._y;
this.OldW = this._width;
this.OldH = this._height;
this.OldMouseX = 0;
this.OldMouseY = 0;

function AssignResizeFunction(mc:MovieClip, xMode:Number, yMode:Number)
{
      mc.onPress          = function() { this._parent.StartResize(xMode, yMode);
      mc.onRelease        = function() { this._parent.StopResize(); }
      mc.onReleaseOutside = function() { this._parent.StopResize();    }
      mc.onMouseMove      = function() { this._parent.OnResize(); }
}

AssignResizeFunction(this.Border1,  0, -1);
AssignResizeFunction(this.Border2,  1,  0);
AssignResizeFunction(this.Border3,  0,  1);
AssignResizeFunction(this.Border4, -1,  0);
AssignResizeFunction(this.Corner1, -1, -1);
AssignResizeFunction(this.Corner2,  1, -1);
AssignResizeFunction(this.Corner3,  1,  1);
AssignResizeFunction(this.Corner4, -1,  1);
this.TitleBar.onPress          = function() { this._parent.startDrag(); }
this.TitleBar.onRelease        = function() { this._parent.stopDrag(); }
this.TitleBar.onReleaseOutside = function() { this._parent.stopDrag(); }

function StartResize(xMode:Number, yMode:Number)
{
      this.XMode = xMode;
      this.YMode = yMode;
      this.OldX = this._x;
      this.OldY = this._y;
      this.OldW = this._width;
      this.OldH = this._height;
      this.OldMouseX = _root._xmouse;
      this.OldMouseY = _root._ymouse;
}

function StopResize()
{
      this.XMode = 0;
      this.YMode = 0;
}
```

```
function OnResize()
{
      var dx = _root._xmouse - OldMouseX;
      var dy = _root._ymouse - OldMouseY;

      if (this.XMode == -1)
      {
            this._x     = this.OldX + dx;
            this._width = this.OldW - dx;
            if (this._width < this.MinW || _root._xmouse > this.OldX + this.OldW)
            {
                  this._x     = this.OldX + this.OldW - this.MinW;
                  this._width = this.MinW;
            }
      }
      if (this.XMode == 1)
      {
            this._width = this.OldW + dx;
            if (this._width < this.MinW || _root._xmouse < this.OldX)
                this._width = this.MinW;
      }
      if (this.YMode == -1)
      {
            this._y      = this.OldY + dy;
            this._height = this.OldH - dy;
            if (this._height < this.MinH || _root._ymouse > this.OldY + this.OldH)
            {
                  this._y      = this.OldY + this.OldH - this.MinH;
                  this._height = this.MinH;
            }
      }
      if (this.YMode == 1)
      {
            this._height = this.OldH + dy;
            if (this._height < this.MinH || _root._ymouse < this.OldY)
                this._height = this.MinH;
      }
}
```

This method has two disadvantages. First, it becomes incompatible with the Adobe Flash player. The second problem is it has many movie clips in it, which generates extra draw primitives and consumes extra memory. This is the price you pay for the simple and general solution. If extra draw primitives are critical, it is possible to combine both methods and replace the borders and corners with a single shape that represents the frame of the window. In which case it is necessary to add extra logic to the ActionScript code, but it should be less complex and less “fragile” than in the first example. The major advantage is that you can use arbitrary shapes for the hit-testing movie clip with irregular borders and corners. Shape and motion tweens will also work.

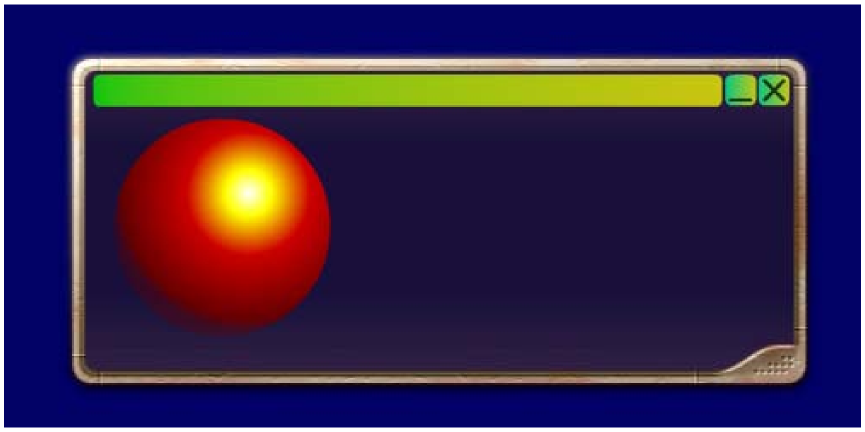
It is important to mention that the Flash Studio has a potential bug when generating SWF files with Scale9Grids. You can reproduce it with scale9grid_window2.fla. If you select the “TitleBar” layer and draw a small rectangle in the center of the window, the Scale9Grid will be incorrect. It appears that the Flash Studio does not tolerate nested movie clips and other graphics in the same layer. In a few cases it even fails to restore the correct Scale9Grid after removing this rectangle. The “Save and Compact” operation may help.
If the problem persists, it is possible to restore the Scale9Grid from the ActionScript. You just trace it while it is correct:

```
trace(this.scale9Grid);
```

If the bug appears and cannot be eliminated, you can restore it using the following code:

```
this.scale9Grid = new Rectangle(24, 35, 363, 138);
```

For an example of a bitmap based background, see scale9grid_window3.fla and scale9grid_window3.swf. In this file the movie clips for the corners and borders are made fully transparent, because their only purpose is to support the hit-test functionality. They just replicate the shapes of the corners in the background image.



As mentioned earlier in the document, in GFx you can use a single image as the background, while in Flash you have to slice and tile it manually. In addition, the corner and border movie clip trick will not work in Adobe Flash Player, since it does not preserve the hit-test transformations correctly. Similarly, rotation and skewing of a Scale9Grid clip will not work correctly in the standard Flash Player. However, with Scaleform GFx we tried to make creation of resizable windows as simple and efficient as possible, while also making sure scale9grid supports transformations correctly.