Autodesk®
Scaleform®

Scaleform 3D

This document explains how to use the 3D features of Scaleform 3.2 and higher.

Author: Mustafa Thamer

Version: 2.04
Last Edited: January 15, 2014

Autodesk®
GAMEWARE

Copyright Notice

Autodesk® Scaleform® 4.3

How to Contact Autodesk Scaleform:

Document	Scaleform 3D
Address	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 Overview

Scaleform® 3Di™ lets developers take their applications UI to the next dimension with 3D Flash rendering and animation capabilities.

Beginning with 3.2, Scaleform has added a simple but powerful set of ActionScript 2.0 extensions which provide basic 3D capabilities. Using the 3D AS2 extensions (_z, _zscale, _xrotation, _yrotation, _matrix3d, and _perspfov), developers can create stunning animated 3D interfaces for menus, HUDs, and in-game UIs.

These 3D extensions can be applied to any movieclip, button or textfield object. Furthermore, developers can add their own 3D rendered objects or streaming video to the 3D Flash interfaces to create a unique next generation experience.

AS3 developers using Scaleform 4 can use the native 3D capabilities of AS3 directly, as an alternative to 3Di. This document describes the use and operation of the 3Di AS2 extensions which are available in Scaleform 3.2 and higher.

2 Similarities to Flash 10

In general, the 3Di support provided by Scaleform is very similar to 3D in Flash 10 and is expected to behave the same way. 3Di is implemented using a basic set of AS2 extensions built into Scaleform.

2.1 Limitations

Similar to the 3D support in Flash 10, 3Di has the following limitations:

1. Depth sorting is not performed and therefore objects may not be drawn in the right order. The drawing order is still specified by layering order by default.
2. Back-face culling is not used. Objects which face away from the viewer are still drawn.

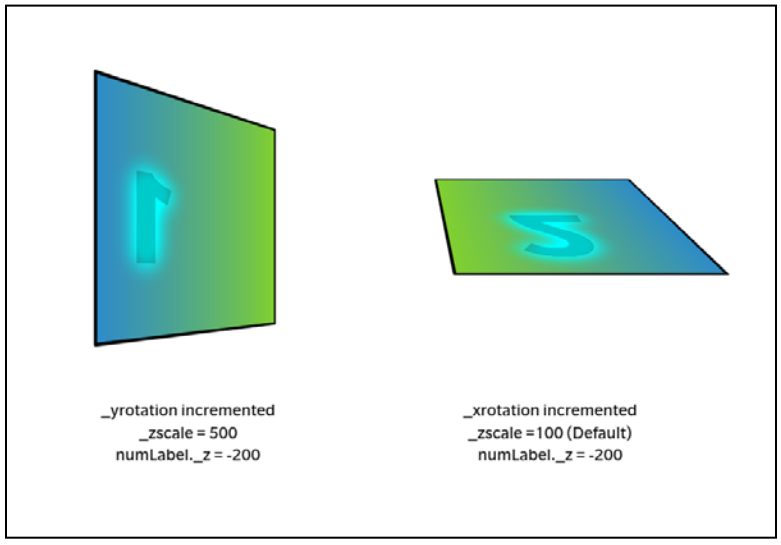

Figure 1: Example of back-facing numbers which are not culled or drawn behind

2.2 Implementation

Scaleform 3Di uses hardware accelerated triangles to render shapes, whereas Flash 10 renders the 3D view into a bitmap and then uses a vector rectangle with the bitmap as a fill for display.

When an object has both 2D and 3D properties, the 3D properties are applied before the 2D properties. This detail is important to understand to get the expected results when rotating an object with the free transform tool as well as 3D operations. Flash 10 has the same behavior.

3 Displaying Flash in 3D

There are two different methods available for displaying Flash content in 3D using Scaleform 4.0 and higher.

3.1 Render To Texture

The first option is to use the traditional method of rendering Flash into a texture and then applying that render texture as a map onto a 3D surface in the scene. Before 3.2, this was the only way developers could visualize Flash in 3D.

Render to Texture method has an advantage over 3Di. Since the Flash content is texture-mapped onto a 3D object, it takes full advantage of the capabilities of the 3D engine and blends seamlessly with the rest of the 3D world. This allows for correct Z (depth) sorting, blending/transparency, and clipping (regardless of the intention to clip the UI).

On the other hand, there are several disadvantages related to performance and memory, since additional memory is required for the render texture. Also, note that any Flash interactivity must be handled by the game (by manually inverse-mapping the mouse coordinates and passing them to Scaleform). Finally, as this approach is texture based, the Flash content will be displayed flat; therefore individual movie clips cannot be offset or rotated from each other. As an example of this approach, refer to the 'SWF to Texture' demo which can be found in the Scaleform SDK Browser.

3.2 3Di

The second option, is to use 3Di, either from ActionScript or the Direct Access API (C++). 3Di has many advantages over using a render texture approach. Firstly, no additional memory is required. Also, individual movie clips can have their own nested transforms – each can be uniquely offset or rotated thus the overall movie does not look flat. Finally, all the input and interactivity is automatically handled by Scaleform, The application simply passes input events to Scaleform as usual.

However, 3Di does have some limitations. Notably, since it is handled by the default Scaleform renderer and treated as a UI, no depth sorting or back-face culling is performed. Therefore as UI objects rotate behind other objects, they may not be drawn in the right order. Also, it may be tricky to seamlessly inter-mix UI elements with other 3D game objects, since the UI objects should match the same camera, lighting, post-processing and other graphical effects in use. Also, many UI elements are transparent and require a specific drawing order for blending, which should be taken into consideration.

4 3Di API

4.1 ActionScript 2 API

With 3Di, Flash objects can now have another dimension. The addition of the third dimension allows users to view objects moving forward or backward towards the viewpoint. As objects move away from the camera, they appear smaller due to perspective projection.

By default, objects are two dimensional and no 3D properties are used or calculated. However, once a 3D property is set (x or y rotation, z translation or scale or 3d matrix) the object becomes three dimensional. At that point, a 3D transformation matrix is automatically created and assigned to the object. The object can be returned to a completely 2D state by setting its 3d matrix to null (i.e. `foo._matrix3d = null` in ActionScript). Setting only the rotations and z values to 0 will not remove the 3D transformation.

As mentioned above, 3D rotation allows rotation around any of the 3 axes, x, y, or z, as opposed to the 2D case which only rotates around the Z axis. Similarly, 3D scale adds an extra axis for scaling, with the _zscale ActionScript extension.

The default 3D coordinate system and camera in Scaleform is the same as the one found in Flash 10. It is a Right-Handed system, with +X to the right, +Y down and +Z going into the screen (away from the viewer). The default camera is looking down the +Z axis with a FOV angle of 55 degrees.

3D properties can be applied to the following types of Flash objects:

- Movie clips
- Text fields
- Buttons

The following ActionScript extensions were added for 3Di:

- **_z** - Sets the Z coordinate (depth) of the object, defaults to 0.
- **_zscale** - Sets the scale of the object in the Z direction as a percentage, defaults to 100.
- **_xrotation** - Sets the rotation of the object around the X (horizontal) axis, defaults to 0.
- **_yrotation** - Sets the rotation of the object around the Y (vertical) axis, defaults to 0.
- **_matrix3d** - Sets the complete 3D transform of the object using an array of 16 floats (4 x 4 matrix). If this value is set to NULL, all 3D transformations will be removed and the object will become 2D.
- **_perspfov** – Sets the perspective Field Of View angle on the object, valid angles must be > 0 and < 180, or -1 if to disable perspective and use an orthographic view. If this value is not set, the object inherits the root FOV angle, which defaults to 55.

The use of these new extensions requires the global variable, `gfxExtensions`, to be set to true in ActionScript.

If you plan on modifying the rotation of a movie clip instance, make sure to note of the location of the registration point. By default, when you create a movie clip symbol, the registration point is set to the top left corner. If you apply a 3D rotation to the movie clip instance using _xrotation or _yrotation, the object will rotate around the registration point. The registration point can be set to the center of the symbol if desired. Make sure to set the registration point carefully if you want to apply rotation to the object using 3Di.

Example 1

Rotate a movieclip 45 degrees around the Y axis:

```
_global.gfxExtensions = true;
mc1._yrotation = 45;
```

Example 2

Continuous rotation around X axis along with a Z scale:

```
_global.gfxExtensions = true;
sq1._zscale = 500;

function rotateAboutXAxis(mc:MovieClip):Void
{
      mc._xrotation = mc._xrotation + 1;
      if (mc._xrotation > 360)
      {
            mc._xrotation = 0;
      }
}

onEnterFrame = function()
{
      rotateAboutXAxis(sq1);
}
```

Example 3

3D transformation using a 3D translation matrix applied to the root:

```
function PixelsToTwips(iPixels):Number
{
      return iPixels*20;
}


function TranslationMatrix(tX, tY, tZ):Array
{
      var matrixC:Array = [   1, 0, 0, 0,
                        0, 1, 0, 0,
                        0, 0, 1, 0,
                        tX,tY,tZ,1];
      return matrixC;
}

this._matrix3d = TranslationMatrix(PixelsToTwips(100),PixelsToTwips(100),0);
```

4.2 3Di from C++

4.2.1 Matrix4x4

A class named Matrix4x4 was added to represent a 4x4, row major matrix. This class has all the appropriate functions for creating and manipulating various types of 3D matrices including general world transformations as well as view and projections. See Render_Matrix4x4.h for details.

4.2.2 Direct Access

The Direct Access interface (via the GFx::Value class) has been expanded to support 3Di. 3D properties can now be set or queried with the new API. These functions are analogous to the 3Di ActionScript extensions.

See the GFx::Value::DisplayInfo class in GFx_Player.h for more details.

```
void    SetZ(Double z)
void    SetXRotation(Double degrees)
void    SetYRotation(Double degrees)
void    SetZScale(Double zscale)
void    SetFOV(Double fov)
void    SetProjectionMatrix3D(const Matrix4F *pmat)
void    SetViewMatrix3D(const Matrix3F *pmat)
bool    SetMatrix3D(const Render::Matrix3F& mat)

Double  GetZ() const
Double  GetXRotation() const
Double  GetYRotation() const
Double  GetZScale() const
Double  GetFOV() const
const Matrix4F* GetProjectionMatrix3D() const
const Matrix3F* GetViewMatrix3D() const
bool    GetMatrix3D(Render::Matrix3F* pmat) const
```

4.2.3 Render::TreeNode

3D matrix calls are added to the Render::TreeNode class interface to set the View and Perspective matrices used for rendering the movie in 3D. Although perspective can be set on an individual display object, it is often convenient to set it at the root of the movie, to affect all objects at once.

`GFx_Player.h`

```
void     SetProjectionMatrix3D(const Matrix4F& m)
void     SetViewMatrix3D(const Matrix3F& m);
```

4.2.4 Example: Changing the Perspective Field of View using Direct Access API

```
Ptr<GFx::Movie> pMovie = …;
GFx::Value tmpVal;
bool bOK = pMovie->GetVariable(&tmpVal, "_root.Window");
if (bOK)
{
      GFx::Value::DisplayInfo dinfo;
       bOK = tmpVal.GetDisplayInfo(&dinfo);
       if (bOK)
       {   // set perspectiveFOV to 150 degrees
           Double oldPersp = dinfo.GetFOV();
           dinfo.SetFOV(150);
           tmpVal.SetDisplayInfo(dinfo);
       }
}
```

5 Perspective

The example code above shows how to set the field of view (FOV) on a movie using the Direct Access API. Perspective and view settings can be applied to the root of a movie or on individual display objects. If an object returns a perspective FOV value of 0 then it will inherit the FOV value from its parent.

Perspective makes objects closer to the viewer appear larger and objects farther away appear smaller. By changing the field of view value, the strength of this effect can be controlled. The greater the value, the stronger the distortion applied to a display object moving along the z-axis. A low FOV value results in very little scaling and a high value gives the appearance of greater movement. The value must be greater than 0 and less than 180, where 1 gives almost no perspective distortion and 179 creates a distorted fish-eye lens effect.

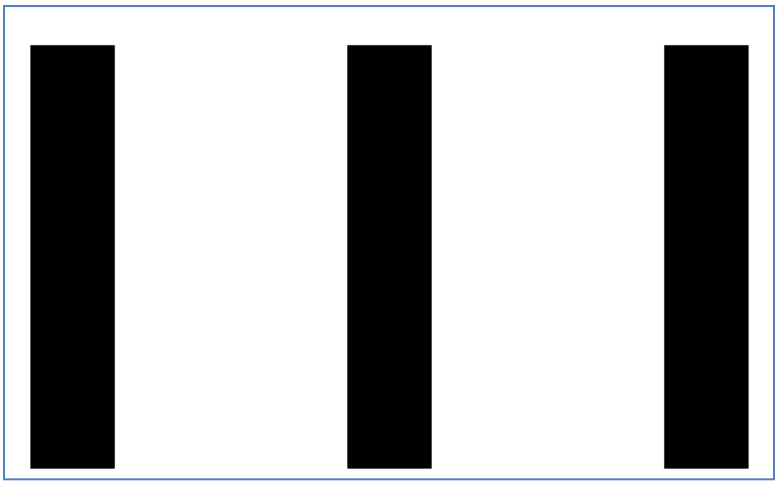

Figure 2: Original (un-rotated) Flash image

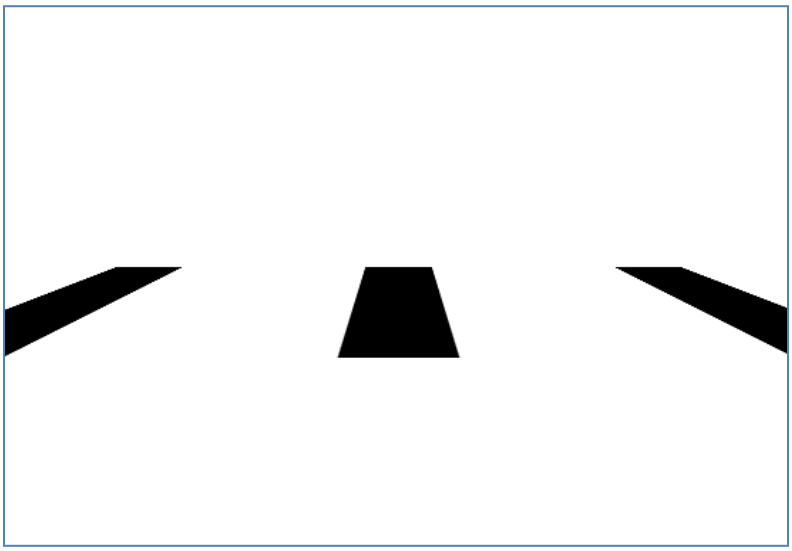

Figure 3: With X rotation of -80 and default perspective (55 degrees)

Low FOV values (below 5) often result in a distorted effect. In order to remove the perspective effect completely, it may be better to set the object to use an orthographic projection. In ActionScript, this can be done by setting the property _perspfov to -1. In C++ it can be done two different ways using the Direct Access API:

// 1. Set Perspective FOV to -1

```
GFx::Value tmpVal;
bool bOK = pMovie->GetVariable(&tmpVal, "root"); //"_root" for AS2
if (bOK)
{
        GFx::Value::DisplayInfo dinfo;
        bOK = tmpVal.GetDisplayInfo(&dinfo);
        if (bOK)
        {
            dinfo.SetPerspFOV(-1);
            tmpVal.SetDisplayInfo(dinfo);
        }
}
```

// or, 2. Set the perspective matrix to an orthographic matrix

```
void MakeOrthogProj(const RectF &visFrameRectInTwips, Matrix4F *matPersp)
{
         const float nearZ = 1;
         const float farZ = 100000;
         float DisplayHeight = fabsf(visFrameRectInTwips.Height());
         float DisplayWidth = fabsf(visFrameRectInTwips.Width());
         matPersp->OrthoRH(DisplayWidth, DisplayHeight, nearZ, farZ);
}

GFx::Value tmpVal;
bool bOK = pMovie->GetVariable(&tmpVal, "root.mc");
if (bOK)
{
     GFx::Value::DisplayInfo dinfo;
     bOK = tmpVal.GetDisplayInfo(&dinfo);
     if (bOK)
     {
         Matrix4F perspMat;
         MakeOrthogProj(PixelsToTwips
                        (pMovie->GetVisibleFrameRect()),  &perspMat);
         dinfo.SetProjectionMatrix3D(&perspMat);
         tmpVal.SetDisplayInfo(dinfo);
     }
 }
```

It can also be done by setting the Projection matrix directly on the movie object:

```
Ptr<Movie>      pMovie = …

// compute ortho matrix
Render::Matrix4F ortho;
const RectF &visFrameRectInTwips = PixelsToTwips(pMovie->GetVisibleFrameRect());
const float nearZ = 1;
const float farZ = 100000;
ortho.OrthoRH(fabs(visFrameRectInTwips.Width()),
fabs(visFrameRectInTwips.Height()), nearZ, farZ);

pMovie->SetProjectionMatrix3D(ortho);
```

5.1 Setting Perspective in AS3

In ActionScript 3 you can use a PerspectiveProjection object to adjust an object's perspective.

Example:

```
import flash.display.Sprite;

var par:Sprite = new Sprite();
par.graphics.beginFill(0xFFCC00);
par.graphics.drawRect(0, 0, 100, 100);
par.x = 100;
par.y = 100;
par.rotationX = 20;

addChild(par);

var chRed:Sprite = new Sprite();
chRed.graphics.beginFill(0xFF0000);
chRed.graphics.drawRect(0, 0, 100, 100);
chRed.x = 50;
chRed.y = 50;
chRed.z = 0;

par.addChild(chRed);

var pp3:PerspectiveProjection=new PerspectiveProjection();
pp3.fieldOfView=120;
par.transform.perspectiveProjection = pp3;
```

6 Stereoscopic 3D

Scaleform 3Di can be used to create a UI for games and applications which employ stereo 3D imaging. Generally, the stereo technique involves enhancing the illusion of depth by creating two images, slightly offset from each other, one for each eye. These two images represent two perspectives of the same scene and allow the brain to perceive depth. Special glasses, possibly in sync with the stereo display device, ensure that the correct image is shown to the correct eye.

Scaleform Scaleform 3Di works well out of the box with stereoscopic 3D systems such as NVIDIA's 3D Vision Kit on the PC. NVIDIA's solution works automatically at the driver level therefore stereoscopy can be enabled without any code changes. This is not the case for other systems, such as game consoles. For those systems, the application would need to call Display twice, one for each eye, and set the transformation matrices to adjust for the parallax shift of left and right eye. Scaleform supports this and provides an example PS3 stereo player called TinyPlayerStereo which can be found in the Scaleform subdirectory here: *Apps\Samples\GFxPlayerTiny\GFxPlayerTinyPS3GcmStereo.cpp.*

6.1 NVIDIA 3D Vision

6.1.1 3D Vision Setup

In order to render a 3Di UI in stereo with the NVIDIA 3D Vision Kit, first ensure that the kit and associated hardware has been installed correctly. Go to the NVIDIA control panel and click 'Enable Stereoscopic 3D' in the Stereoscopic 3D section. Setup can be verified by running the Stereoscopic 3D Test from the NVIDIA control panel.

6.1.2 Launch

Next, launch your application or the D3D9 Scaleform Player in full screen mode. The automatic 3D Vision support only works in full-screen, and requires the Direct3D Scaleform Player, therefore the application must start up in full-screen mode. In the case of Scaleform Player, this requires passing in the command line argument –f. Depending on the settings in the stereo section of the NVIDIA control panel, the application may start up with stereo 3D disabled. If the 3D effect does not appear immediately, hit (CTRL + T) to toggle stereo mode.

6.1.3 Convergence Profile

Once the stereo effect is active, it is important to adjust the settings. Objects at screen depth should have zero separation and no stereo effect. Similarly, objects which are relatively closer and farther away must have the appropriate positive and negative separation to appear out of or in to the screen. Set the 3D Vision

depth and convergence values that look best for the application and then save them in a registry profile for future use.

Normally when using 3D Vision only control over the level of depth is possible, by either turning the scroll wheel on the IR transmitter or by using (CTRL + F3) /(CTRL + F4) to decrease or increase it. This is normal, since for most games the convergence does not require adjustment because everything has been set in the game's profile. For custom applications or when using Scaleform Player, the correct convergence must be set the first time. The default keys to change the convergence level are (CTRL + F5) and (CTRL + F6), however note that they are not active by default. Follow the steps below to activate this feature.

In order to control the level of convergence one must first "Enable the advanced in-game settings" from NVIDIA's Control Panel by going to "*Stereoscopic 3D*", choosing "*Set up stereoscopic 3D*" and then opening the "*Set Keyboard Shortcuts*". When the advanced settings are enabled, the convergence can be changed using (CTRL + F5) and (CTRL + F6) and also use the (CTRL + F7) key combination in order to save the selected custom settings. Saving the settings allows users to skip over the convergence configuration step the next time the application is run.

When adjusting convergence, unlike depth there is no graphical representation on-screen showing the convergence level. Therefore when changing convergence one must carefully watch the changes on the screen. The best way to proceed is to view a scene where an object should be at the screen depth (or perhaps a middle/reference depth) and then hold down (CTRL + F5) until the image separation starts changing. This can take 10 to 15 seconds as the convergence is being adjusted in very small increments and will not be noticeable at first. Hold down (CTRL + F5) or (CTRL + F6) until the object at screen depth appears to have no separation of images (when viewed without glasses). This will set the convergence to 0 using the object at screen depth as a reference point. Again, once the convergence and depth amounts selected are deemed correct, pressing (CTRL + F7) will save those settings to the registry for the application to use in the future.

NVIDIA has provided an API to access and control stereo 3D behavior programmatically, as well as some documentation on best practices when designing a game for stereo 3D. Please see the links below for more detail:

API : http://developer.nvidia.com/object/nvapi.html.

Designing for stereo
http://developer.download.nvidia.com/presentations/2009/SIGGRAPH/3DVision_Develop_Design_Play_in_3D_Stereo.pdf.

6.2 The Scaleform Stereo API

For consoles, Scaleform 4 includes a simple API in Render::HAL, for supporting stereo 3D. It is up to the application to initialize the hardware for stereo 3D and set up the frame buffers and surfaces to support HDMI 1.4 frame packing.

Initially, the application should initialize the stereo parameters it wants in the Scaleform renderer, as shown in Section 6.2.1

Scaleform 4 provides the ability to set the viewport to render separate stereo images side by side or top and bottom. This is a convenience for devices which require stereo images to be tiled vertically or horizontally.

This can be done by calling Render::Viewport::SetStereoViewport with the appropriate flag.

Here are the new Viewport Stereo option flags that were added to SF4:

```
        // For stereo in display hardware only; uses the same size buffer but half
           for each eye.
        View_Stereo_SplitV        = 0x40,
        View_Stereo_SplitH        = 0x80,
        View_Stereo_AnySplit    = 0xc0,

    void SetStereoViewport(unsigned display);
```

6.2.1 Scaleform Stereo Initialization

```
Render::StereoParams s3DInfo;
s3DInfo.DisplayDiagInches = 46;         // 46 inch TV
s3DInfo.DisplayAspectRatio = 16.f / 9.f; // widescreen aspect ratio
s3DInfo.Distortion = .75;               // 0 to 1 value
s3DInfo.EyeSeparationCm = 6.4;   // this will default 6.4cm
pRenderHAL->SetStereoParams(s3DInfo);
```

During the display traversal, the application should call Display twice, once for each eye, and take care of switching to the correct frame buffer surface before each Display call. Scaleform will handle offsetting the view for each eye. For example:

```
pMovie->Advance(delta);

pRenderer->GetHAL()->SetStereoDisplay(Render::StereoLeft, 0);
pMovie->Display();
```

```
pRenderer->GetHAL()->SetStereoDisplay(Render::StereoRight, 0);
pMovie->Display();
```

The second parameter in SetStereoDisplay indicates whether the drawing surface should be changed or not when rendering stereo images. If set to true (or 1), the render target is reset.

7 Sample Files

An excellent way to experience 3Di in action is to take a look at some of the 3D sample Flash files (SWFs and FLAs) that are provided with the Scaleform SDK.

Sample files are located in the following directory by default: *C:\Program Files (x86)\Scaleform\GFx SDK 4.3\Bin\Data\AS2 or AS3\Samples*

Figure 4: 3DGenerator

Figure 5: 3D Inventory

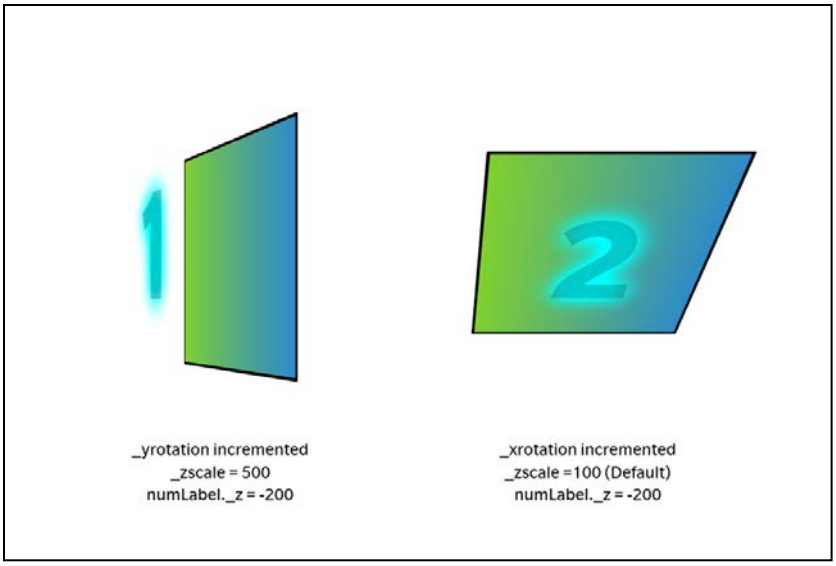

Figure 6: 3D Squares

Figure 7: 3D Window