# FLASH&FLEX

DEVELOPER'S MAGAZINE

Vol.4 No.11 Monthly Issue 11/2011 (29) November ISSN 1898 -9136

# Crocus Modeller Fresh UML tool for Flash Platform!

## INTERVIEW WITH LAURENT HASSON!

## APPLIED DESIGN PATTERNS SINGLTETON

## CROCUS MODELLER REVIEW

## CREATING A STATEMENT OF WORK FOR YOUR FLASH/FLEX PROJECTS

## BLACKBERRY'S FATHER ABOUT BRAND NEW TOOL ALICE

## STARTING ACTIONSCRIPT PROGRAMMING – PART TWO

## + ADOBE AT BLACKBERRY DEVCON AMERICAS 2011

# STREAMING + INTERACTIVITY = LIMITLESS POSSIBILITIES.

The new FMS 4 – Flash® Media Interactive Server (FMIS) & Flash Media Enterprise Server (FMEN) – offers better performance in interactive media streaming.

## Adobe Flash® Media Server 4 Advantages:

- New RTMFP Multicast P2P capability
- Reduces latency and bandwidth usage
- Interactive access to buffered media
- Faster switching with RTMP Dynamic Streaming

FMIS plans starting at $19.95/mo.
FMEN pricing, please contact us.

INFLUXIS

www.influxis.com

# STREAMING + INTERACTIVITY = LIMITLESS POSSIBILITIES.

Our new mobile streaming lets you reach your audience on the go.
Deliver your videos to smartphones and tablets with ease.

**Mobile Streaming Advantages:**

- For VOD or live streaming
- Intuitive and easy-to-use
- Optimized for all mobile platforms and social media
- HTML5-ready
- Custom mobile streaming solutions

Start with a free account.

INFLUXIS

mobile.influxis.com

# Stage3D

## Opens The Doors To a New Gaming Era On The Internet

It has been a long time since Adobe showed off at MAX 2010 their Stage3D technology. Now, we are just off with an official release in early October (October 3th to be precise), and it has Stage3D have included in Flash Player 11 and AIR 3 client technologies.

### What you will learn…

- You will get acquainted with the new Flash capabilities for 3D animation and how the 3D rendering pipeline works. You will learn some things about 3D representation in the way. This will become the foundation to understand more complex projects.

### What you should know…

- You should be familiar with programming as3 with Flash Professional or Flash Builder.

Now this might raise a couple of questions: What is this Stage3D? You might remember it under the *Molehill* codename. Why should I care? Well, as you continue reading I believe that you will agree with me that this is the new kid on the block. And, of course, it was one of the show stoppers at Adobe MAX 2011.

Stage3D represents Adobe stake at Flash technologies in the entertainment market, which continues to grow in revenue each year. So, it is very big news. With new features added to the Flash architecture, it will surely be a thriving year for all Flash developers. And it is a great way to fight all this campaign against Flash, with people challenging with doubts over the Flash platform future. Now that the scenario looks a lot steady, we can start looking into the future. And Stage3D is a big piece of it.

First of all, I will have to mention the 3 big pieces that Adobe has put into Flash Player 11 and AIR 3:

- A modern architecture. Yes, we finally got to see the end of the 64 bit tunnel. This one was a really expected feature, and it seems that we all get to install the 64 bit player without using the labs version. This means (I believe) a normal update path.
- Flash everywhere seems nearer. With the adoption of iOS, Android, OS X, and Windows native extensions, now the capabilities of the client are not limited to the existing libraries included. Now we will be able to develop native code libraries of our own.
- Captive is no longer captive of iOS. We can produce and install stand-alone apps on Android, Windows and Mac OS. Of course, there are some limitations to it, which are similar to Adobe Director (we are not able to produce a Mac app from Windows, and vice versa).


**Figure 1.** *64 bit support finally makes it into Flash*

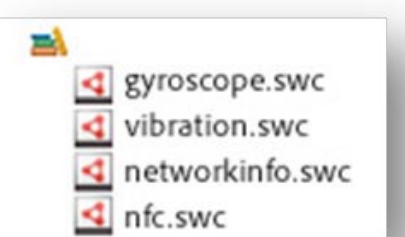



**Figure 2.** *Native extensions leverage devices capabilities*

**Figure 3.** *Captive means more platforms with ease*

So, by mashing up all this incredible features with the Stage3D API that incorporates native GPU acceleration, we are having a new old friend contending in the gaming market, once again. Game producers are now able to deliver games in a very broad ecosystem that includes PC, tablet, smartphone and connected TV. All in a single platform, which lessens costs and increments market reach. But non-gaming ActionScript programmers also get benefited with it, since they will be able to leverage their existing knowledge and libraries and explore this area.

Related, but still apart, another interesting area that will be empowered with these technologies is the serious gaming field. By joining gaming knowledge with business and social programmers, Flash will be able to become the most important platform for social non entertainment gaming.

Adobe is also continuing with its already successful media effort. Counting aboard with Amazon, ESPN, HBO, Hulu and YouTube as some of its customers, Adobe is putting the Flash platform at the center of everything named media. The new HD video accelerated by hardware assures smooth playback, and Adobe Pass gathers even a larger crowd through their TV Everywhere implementation.

The business market will also benefit from data driven apps that Flash has always been famous for, but with a broader audience, and better on screen animations that help understand business process and facts better. This way, Flash might become the ultimate communication and collaboration tool. With bigger and better access from everywhere to data sources and a more fluent visual language, it will be a great tool to help on decision making tasks.

Stage3D in conjunction with the new features and improvements, nearing its release time, will break free the gaming console experience into other markets. Adobe comments on its Flash platform blog that 70% of web games, 9 of the 10 top games at Facebook and 70% of the games on Google+ are powered by Flash. Zynga and EA are also big players in the social arena, and both are using this platform. So the 3D experience liberated by Stage3D will be able to reach a market that is 11 times larger than the one of the biggest selling game console.

Stage3D comes along with dynamic audio (they are promising a mature audio platform), high quality voice chat, low latency peer to peer multiplayer networking, full screen experience (including full HD 1080p video playback), and native support for mouse, multi touch devices and camera inputs. So it indeed smells and feels like a gaming console, from hardware to screen to user experience. And there seems to be a lot of surface to cover.

With hundreds of millions of installed clients capable of upgrading into these new possibilities, the market is open and first comers are first saved.

But of course, having such a fantastic set of features could not be done using the old architecture, so talking about Stage3D, we are going to see many new concepts introduced and we will see many different styles of implementation through the different engine frameworks that make the best of this new GPU accelerated surface that is Stage3D. Stage3D is able to accelerate 2D a 1000 times. That is a lot of new power, and it can enable some really interesting new applications, not only in the entertainment area.

Stage3D is designed to make use of the GPU and other characteristics found on mobile, and it is designed with mobile in mind. We won't get a fully featured PC native player (not a real limitation if you just remember there are the native extensions), but this guarantees us that we are not going to see a limited Flash Player on devices, and finally we will get a more standardized user experience across all platforms. And this gets more exciting if you take into account that with the actual web


**Figure 4.** *Adobe Pass platform uses Flash extensively*

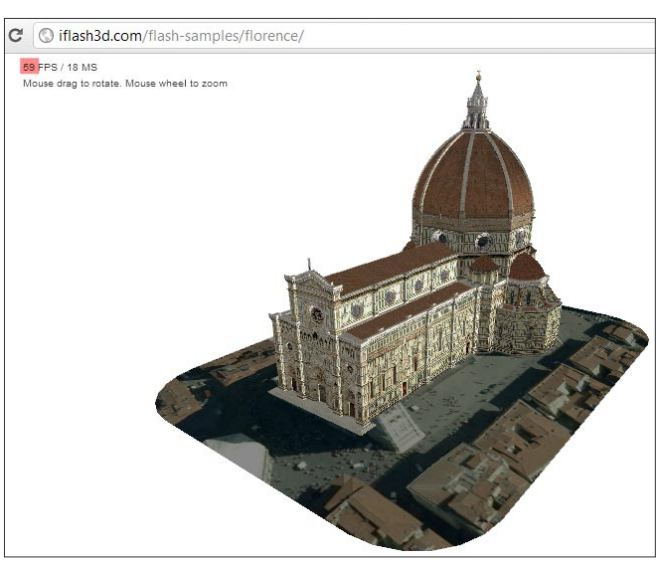



**Figure 5.** *Stage3D makes new kinds of applications possible*

upgrade rhythm we could expect 50% of these users to upgrade, and it will be the biggest GPU accelerated platform on the web in a very short time. So, for some of us we will have a very big client surface by the time we release a product based on Stage3D architecture.

There is another piece to this puzzle. We are expecting to see if the preview release of the Adobe 3D framework codenamed as *Proscenium* will ever see the light as an official code library. This resource is intended to help Flex Framework developers to have a head start into the world of Stage3D. It is aimed as a showcase of the current capabilities of the Stage3D API.

It is not meant to compete with more complete game engines and frameworks, but it is pretty useful on making simple high quality rendering of single objects or small collections, contemplating the lighting, shading, reflection and shadow attributes. Why is it so useful? Well, I find particularly useful the support of COLLADA and OBJ formats, as well as functions that implement common object creation (flat surfaces as squares, and volume objects as spheres and boxes). There is also an interesting support to spline based and scripted animations, as well as bone animation, so simple projects can go up with a small amount of effort.

I will talk about Proscenium on a following article, as well about some other interesting topics. But for now, I will now discuss some basics that are behind the Stage3D API so we can understand in a better manner the way things are worked through the 3D pipeline. This knowledge will be very useful even if you pretend to use a higher level framework.

There have always been very good open source and commercial 3D libraries that help us incorporate attractive effects into our projects. But nonetheless, we have always felt the limitations of such libraries, which after all had to use the power of our CPU to process the entire 3D pipeline. There has been a tremendous advance in the 3D applications in modern day computers which make way big differences, especially in games. The configuration item which is responsible for this is the GPU.

The CPU (*Central Processing Unit*) is the general purpose processor that sits at the heart of our computers. It is in charge of all the processing that goes in every piece of software that runs in our computer, and that includes the Operating System. It is also responsible of transferring enormous quantities of information between the hard drives and the volatile memory. These activities are continuously happening in our systems. But most important, it is prepared for many kinds of processing, but is not good on a particular kind, so when we delegate 3D processing into it, it has a less than optimal performance. On the other side, and sitting in our graphics cards is the GPU (*Graphics Processor Unit*), which is a specialized kind of processor, which knows how to do less things than the CPU, but it does tremendously well. And 3D processing is the kind of work for which they are optimized for. Besides that, the GPU is free of making any other kind of tasks, like executing the Operating System or transferring memory chunks. This is why GPU acceleration improves 3D graphics performance.

Flash Player 11 is a GPU accelerated software, so when we run 3D on it, it has now the power given by the GPU in modern computers. But it falls smoothly to CPU processing if a GPU is not found (or drivers are not he adequate ones). But, the differences between both processing processes are enormous.

To understand the differences, we will explain a little more about how 3D images are created in our computers.

The final image that we see as a 3D visualization is made through a series of steps that has been standardized as a pipeline. On the older way of programming 3D graphics, these steps were made in what is called a fixed pipeline. This type of pipeline transforms the data clouds that describe 3D objects into a rendered image.

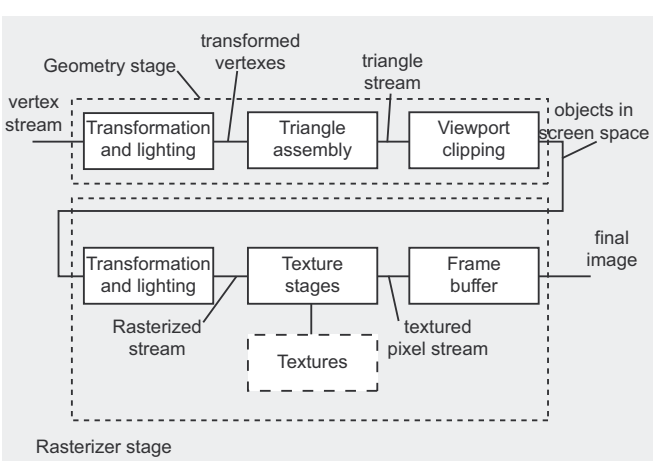



**Figure 6.** *The fixed pipeline is the classic 3D rendering model*

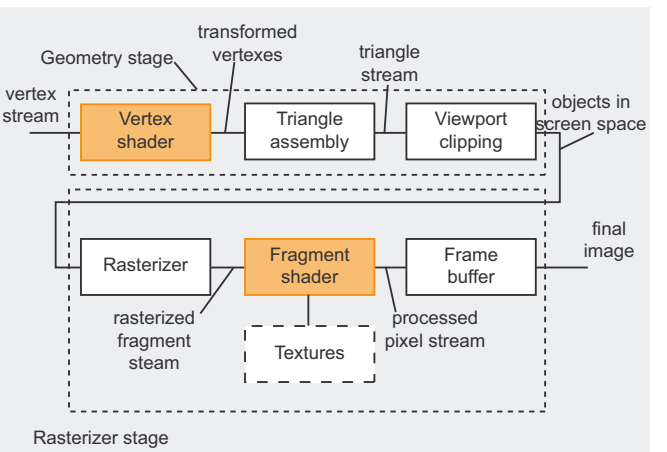



**Figure 7.** *The introduction of programmable shaders improved the image quality*

3D objects are made of several triangles, which are described in space through their vertexes. With interactive 3D animation we use triangles, because it is the only polygon that always stays flat, no matter how you move each of their vertexes in space. This is useful for graphical representation, since it allows us to represent illumination, according to the angles between the triangle, the light source and the viewer. It is also important to say that these polygons are 1 sided. So it is only possible to view one side of them. The other one is transparent, but we can also have a 2 sided polygon if we duplicate the lighting calculations, which can increase the processing power demands easily. If, for example, we create a 2 sided object with 1 million triangles, it would count as 2 million triangles, which might fall easily off . So, we should be careful when turning 2 sides on our triangles when creating 3D objects, if those extra faces are not necessary. With Stage3D we control it with the setCulling method in the Context3D object.

Turning back to the fixed pipeline, it is described by Marco Scabia on its *How Stage3D Works* article (*http://www.adobe.com/devnet/flashplayer/articles/how-stage3d-works.html*) on the Adobe Developer Connection site as the following sequence of steps: Figure 6.

This pipeline is quite simple. It just needs the data cloud (our 3D objects and their position), some texture data, lights and the camera position to render the final image.

Marco Scabia also describes in his article the following programmable pipeline, which is the result of the GPU evolution: Figure 7.

The main differences between these two pipelines are the introduction of programmable blocks inside the pipeline. And although these seem to be small changes, they really make a huge difference on the quality of the final image.

However, we are also warned that there is a price to pay for the added quality and speed of the final rendered image, and that is that we are now in charge of programming our own Shaders. But, do not worry; we will be covering this topic soon (but not in a detailed manner in this article) when we deliver our first simple scene.

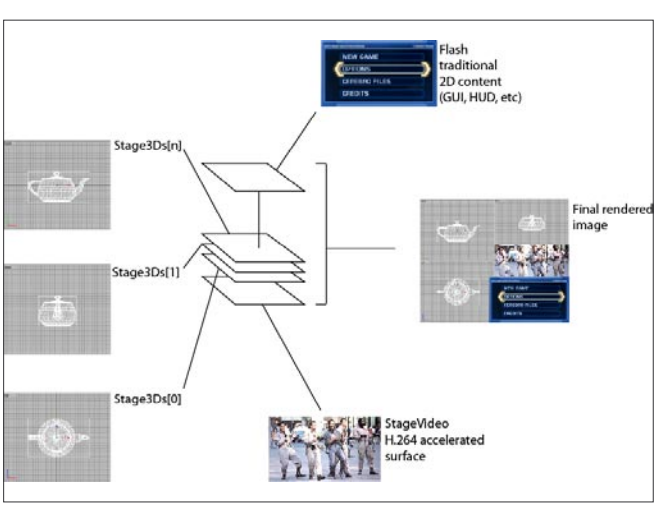



**Figure 8.** *StageVideo, Stage3D and Stage stack*

Before Flash Player 11, a 3D library was able to render up to 4,000 triangles before degrading the experience with lags and other problems visible problems. En even then, the visual quality was missing some graphic details that we are used to in modern games and visualizations like glows, blur, and reflections. This kind of visualization on a GPU accelerated computer is able to come out with up to a million triangles being rendered to screen.

Now, we have more control over the vertices, which are part of our objects, and over the pixel colours of our image through Vertex Shaders and Fragment Shaders, respectively.

Stage3D has the advantage of being ActionScript 3 based. This means that you are not facing specific details for each type of hardware in which you want your application to run. But, there are also some disadvantages if you are trying to make use of every last resource available on a specific platform. So, at the end it is just a matter of power versus reach.

I consider that the power of having a 2D Flash based interface running along with my 3D application is a very useful feature that one could make use of in multiple situations. This kind of scenario is not always easy to solve. I believe that the stage paradigm, along with the mixture of 3D with classic Flash animation, is somehow a reminiscent of Adobe Director. Somehow, Flash technologies are now coming slowly to cover the empty space that Director had left. And that, along with a huge platform reach, makes a next generation platform, worth exploring. Now, it is time to begin our exploration.

Flash is built on top of a Stage and DisplayObject architecture. Each DisplayObject is added to the Stage in order to be displayed. Stage becomes the root container of everything on screen. Stage3D is just a new kind of Stage, dedicated to 3D.

So we now have a new stack order. First, we have at the bottom of the stack a StageVideo object, which is dedicated to high performance video playback with the benefits of H.264 hardware accelerated decoding. This object was introduced in Flash Player 10.2. It allows us to present video inside a texture sitting behind the stage and being painted completely by the GPU.

After the StageVideo, you may place several Stage3D layers. This has the advantage of allowing a viewport paradigm for your applications. For example, you could have several rectangular viewports on different parts of the screen showing different views. There is no way to blend or mix different Stage3D objects, so even if you could overlap them, you would get no benefit, unless it

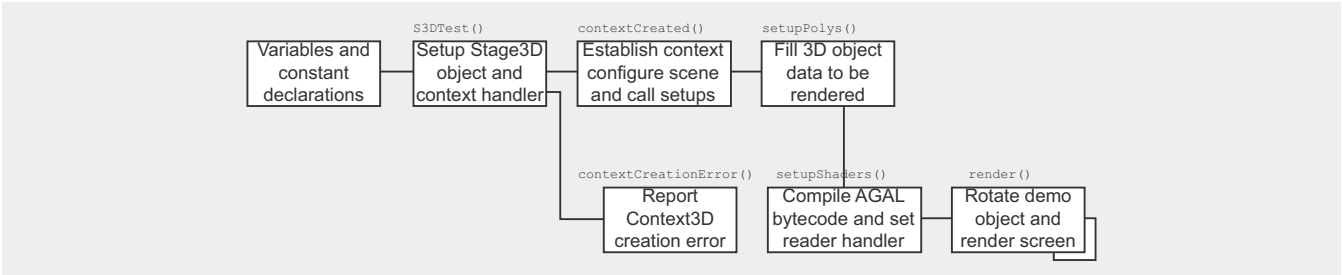



**Figure 9.** *Article program flow*

is part of your UI strategy, since you will always see only the one on top.

On top of the stack, you place the Stage that you have always known with 2D capabilities. 2D content can overlap 3D content easily, but it does not work the other way around.

Before we start on our journey, I found a small problem. I am using Adobe Flash Professional CS5.5 (11.5.1.349) and it has all updates sent from Adobe. I assumed I was going to start coding for FP11, but it was not that simple. I later found that there should be an update coming (including Flex 4.6 that will have native FP11 support).

First, I had compiler errors stating *flash.display: Stage3D not found* which got me wondering. But I found a solution searching on the Internet. It still is not clear to me, why Adobe cannot solve this simple thing, or if there is something wrong with their update process. Although, I believe that the problem is that they assume that people has configured their software since the time they published Molehill on their Adobe Labs. It also came to my mind that maybe they have not released yet an updated Flash Professional to support new features in a proper manner. I noticed that the internal Flash Player in Flash Professional is still 10.2. So, if you are planning to follow along this tutorial, please have in mind that you should always test your movie under Debug Mode.

Anyway, if you have the same problem I had, this is the workaround solution:

Download *playerglobal11_0.swc* from the Adobe site (*http://www.adobe.com/support/flashplayer/downloads.html*) and copy it into a new folder named FP 11

**Listing 1.** *Changes needed to develop for FP11*

```
<player asversion="3" id="FlashPlayer11"
                    version="13"></player>
<playerdefinitionpath as2="$(UserConfig)/Classes/
FP10;$(UserConfig)/Classes/FP9;$(UserConfig)/Classes/
FP8;$(UserConfig)/Classes/FP7" as3="$(AppConfig)/
ActionScript 3.0/FP11/playerglobal.swc">
 </playerdefinitionpath>
```

**Listing 2.** *Code schematic*

```
package{
        // import calls
        class S3DTest(){
                // define constants and variables
                function S3DTest(){
                        // Get the Stage3Ds[0] object
                        from Stage
                        // Request Context3D object and
                        its event handler
                }
                function contextCreated(){
                        // Setup Context3D object
                        // Call polygon and shader setup
                        sections
                }
                function setupPolys(){
                        // Fill the vertex and triangle
                        data sets
                }
                function setupShaders(){
                        // Compile AGAL code and upload
                        the Program3D
                        // Setup camera and object
                        initial position
                        // Declare a frame event handler
                        for animation
                }
                function render(){
                        // Render loop
                        // Rotate object and sum up
                        transformations
                        // Render screen
                }
                function contextCreationError(){
                        // Output error information
                }
        }
}
```

in your Flash installation directory (on Windows it is normally *C:\Program Files (x86)\Adobe\Adobe Flash CS5.5*) under *Common>Configuration>ActionScript3.0* . Remember to create a new folder named FP11 and rename the file as *playerglobal.swc*.

In the same installation directory, under *Common >Configuration>Players*, you will find several xml definition files. Copy the latest one and edit the file in a text editor. You will have to make 2 changes: Listing 1.

Basically, you should change the version to 13 (that is the internal version for FP11) and the player definition path to point to the directory where you copied the new *playerglobal.swc*.

With that configuration set into place, now you can set in your Publish Settings the Flash Player 11 as a target. That will do the trick to recognize the new 3D objects.

You should also copy the release and debug versions of the Flash Projector under the installation directory in the Players folder. With that out of the way, we can continue.

The main 2 objects that we will be using are Stage3D and Context3D. Stage3D will be the canvas where our 3D scene will get represented, but Context3D is the render surface that contains all the methods and properties necessary for the rendering operation. It will be then the main class that we will be using.

In a typical workflow, we first ask the Flash Player for a 3D context. We get a Context3D object as a result. If for some reason, the GPU is not supported or the driver is incompatible, we still get a Context3D, but now a CPU based one. Although, it is not an accelerated context, it is still based on a very fast rasterizer algorithm. It is based on SwiftShader from TransGaming Inc. So, you could still expect a 10x performance improvement against older Flash Player versions.

Since Stage3D is based on a programmable pipeline, we will have to write our own vertex and fragment shaders, as was mentioned before. These have to be programmed in 2 ways. The first one is on the low level language AGAL (*Adobe Graphics Assembly Language*) bytecode as a ByteArray. The second one is on a higher level language like Pixel Bender 3D (which is currently in beta stage on Adobe Labs).

Going through the first part of the pipeline we meet also the VertexBuffer3D and IndexBuffer3D objects. The Vertex Buffer will allow us to describe each individual point in space. The Index Buffer will allow us to group these individual vertices in triangles. It is important to note that the Vertex Buffer is a list of attributes per vertex. That is, we are allowed not only to pass the spatial coordinates, but also additional information like vertex colors, or UV coordinates for texturing.

In the big picture, after having the vertex shaders and fragment shaders, we will use the Program3D object to upload our data into the graphic card.

In order to work with AGAL, we will need to download (*http://www.bytearray.org/wp-content/projects/agalassembler/com.zip*) the `AGALMiniAssembler`, which allows us to compile AGAL assembly language into the necessary bytecode. This library is not official yet, but everybody says that it is almost release ready. As mentioned before, Pixel Bender 3D application is another other option to generate the necessary shader code.

**Listing 3.** *Constants for camera physical attributes*

```
public const viewWidth:Number=640;
public const viewHeight:Number=480;
public const camZNear:Number=1;
public const camZFar:Number=500;
public const camFOV:Number=45*Math.PI/
       180;
```

**Listing 4.** *Variables related to Stage3D technologies*

```
private var myS3D:Stage3D;
private var myC3D:Context3D;
private var myIBuffer:IndexBuffer3D;
private var myVBuffer:VertexBuffer3D;
```

**Listing 5.** *Declaration of 3D objects on the scene*

```
private var screenProjection:
       PerspectiveMatrix3D=new
       PerspectiveMatrix3D();
private var objectMatrix:Matrix3D=new
       Matrix3D();
private var cameraMatrix:Matrix3D=new
       Matrix3D();
private var transformMatrix:Matrix3D=new
       Matrix3D();
private const pivot:Vector3D=new
       Vector3D();
```

**Listing 6.** *AGAL code strings*

```
private const VERTEX_SHADER:String="m44
       op, va0, vc0 \n"+"mov v0, va1";
private const FRAGMENT_SHADER:
       String="mov oc, v0";
```

**Listing 7.** *Variables that describe objects on scene*

```
private const dataPerVertex:int=6;
private var vertexes:Vector.<Number>=Vec
       tor.<Number>([0,0,0, 1,0,0,…]);
private var indexes:Vector.<uint>=Vector
       .<uint>([2,1,0,3,2,0,…]);
```

When I was browsing through Adobe's ActionScript 3.0 documentation, I found a nice example to start with. I have made some changes on the code to make it easier to explain each part of it, and I have also commented the code almost line by line.

The logic behind the 3D workflow goes around the cited programmed pipeline.

As a code structure, the sketch is represented by the following lines: Listing 2.

There are 2 important libraries used on the code: `com.adobe.utils.AGALMiniAssembler` and `com.adobe.utils.PerspectiveMatrix3D`. These libraries are not yet an Adobe official release, but they may be soon. The `AGALMiniAssembler` library allows us to convert the low level shader code into the bytecode necessary. Remember that there is also the Pixel Bender 3D option. The `PerspectiveMatrix3D` will help us to control camera perspective related properties, such as field of view and 3D space clipping.

The code starts declaring a VGA screen size for our 3D animation to run in. It also defines the Z space. Imagine that there is a cube behind your computer screen, with the frontal face aligned with the physical screen of your monitor. X space runs from left to right on our model, Y runs from the bottom to the top and Z runs from your computer screen and away from you. camZNear and camZFar describe how far in this 3D world coordinates your sight can go. The more far you allow users to view, the more memory that is consumed.

The camFOV describes the focal aperture of your lens. The more aperture you have on your lens, the more pronounced perspective effect you get.

The most important objects for 3D in Flash are the Stage3D and the Context3D. Everything is done at the Context3D object. It represents the GPU accelerated surface, while the Stage3D represents the screen space where the context will be rendered. Remember that you have many Stage3D layers between the StageVideo and the Stage elements.

**Listing 8.** *AGAL related variables*

```
private var myVertexAssembler:
        AGALMiniAssembler = new
        AGALMiniAssembler();
private var myFragmentAssembler:
        AGALMiniAssembler = new
        AGALMiniAssembler();
private var myProgram:Program3D;
```

**Listing 9.** *Constructor function*

```
public function S3DTest():void{
        this.stage.scaleMode =
        StageScaleMode.NO_SCALE;
        this.stage.align =
        StageAlign.TOP_LEFT;
        myS3D = stage.stage3Ds[0];
        myS3D.x = 10;
        myS3D.y = 10;
        myS3D.addEventListener(
        Event.CONTEXT3D_CREATE,
        contextCreated );
        myS3D.addEventListener(
        ErrorEvent.ERROR,
        contextCreationError );
        myS3D.requestContext3D(
        Context3DRenderMode.AUTO );
}
```

**Listing 10.** *Context creation event handler*

```
private function contextCreated( event:
        Event ):void{
        myC3D = Stage3D( event.target
        ).context3D;
        myC3D.configureBackBuffer(
        viewWidth, viewHeight, 2, false
        );
        myC3D.setCulling( Context3DTrian
        gleFace.BACK );
        setupPolys();
        setupShaders();
}
```

**Listing 11.** *3D object setup function*

```
private function setupPolys():void{
        myVBuffer = myC3D.createVe
        rtexBuffer(vertexes.length/
        dataPerVertex, dataPerVertex);
        myVBuffer.uploadFromVector(
        vertexes,0,vertexes.length/
        dataPerVertex);
        myIBuffer = myC3D.createIndexBuf
        fer(indexes.length);
        myIBuffer.uploadFromVector(index
        es,0,indexes.length);
        myC3D.setVertexBufferAt(0,myVBuf
        fer,0,Context3DVertexBufferForma
        t.FLOAT_3);
        myC3D.setVertexBufferAt(1,myVBuf
        fer,3,Context3DVertexBufferForma
        t.FLOAT_3);
}
```

The VertexBuffer3D object is dedicated to contain a collection of 3D vertexes with their attributes. These are not enough to describe a 3D object, and that is what VertexBuffer3D object helps to achieve. VertexBuffer3D object holds triangle definitions by grouping vertexes by 3. So both objects help describe fully a 3D object.

There are 3 transformations that can be performed to any object: translation, rotation and scaling. Translation describes the actual position of an object relative to the last position. The same goes for rotation and scaling (apparent size relative to world coordinates). These operations are made in linear algebra and its results are stored in matrixes. We need 4 matrixes and 1 vector for our example. A PerspectiveMatrix3D object that will contain data relative to the optical properties of the camera. We need 3 Matrix3D objects to represent the position, rotation and scaling for each object in the scene: our 3D object, the camera (which represents our virtual position in 3D space) and a final matrix named transformMatrix that will accumulate all transformations and give the final position of things.

This accumulation works this way. If you move your object 3 units to the back, and advance the camera object 3 units in the same direction, as you might guess these movements would be accumulated on the transformMatrix and in the screen there would be no apparent movement, since one movement cancels the other. That is, the camera would be at the same distance. If you move the object 3 units and the camera only 1 unit, the difference of 2 units would be the effective object movement on screen.

The Vector3D object is used as an auxiliary pivot (such as an empty node in a 3D program) to be used for rotations on this program.

As you remember, there are 2 shaders that are needed to complete the programmable render pipeline. The Vertex Shader describes any operation that is made to each vertex, before passing this data to the GPU renderer. For example, this might be used by bones animation, displacement mapping, or any other effect you want over the vertexes of your polygons. The Fragment Shader has effect on the pixel level. This way you can alter the final look of the image, by processing colors, illumination and others.

Each shader is described in AGAL on our example. Basically, what is done on the Vertex Shader is that we just process the vertex data by simply passing the initial position and color to the renderer. The Fragment Shader does the same.

We declare the size of our vertex data with a length of 6, because we will put our data formatted in that way into our vertexes Vector. 3 elements will be for x, y and z coordinates. The other 3 elements will represent r, g and b colour values (in a 0 to 1 range).

The indexes Vector will contain our triangle data, so in this example, the first triangle is made out of the vertex elements with a position of 2, 1 and 0 at the vertexes Vector.

We need 2 assembler objects to process and store the Vertex Shader bytecode, as well as a Program3D object to upload the shader programs to the pipeline.

The S3DTest is the main function that starts it all. It is the constructor. We first make some standard Flash routines, like establishing some basic Stage properties.

Here we take our Stage3D variable, named myS3D. We reference stage.stage3Ds[0] we are referencing the bottom 3D layer from the Stage object. The position of this screen space is 10 pixels down from the top and right from the left margin in the Stage area.

At the end of the main function, the event handlers for the success or failure of the Context3D object request are defined, and the Context3D is requested.

Once we have declared the event handler and requested the Context3D object, we should take some things into consideration. The `CONTEXT3D_CREATE` event may be fired at any time, so we must take into account that this function is not a one-time player.

With the `configureBackBuffer` method of the Context3D object, we setup the video buffer with an antialiasing level of 2. Not so good looking, but it won't demand too much processing time for the GPU. You can set this according

**Listing 12.** *Shader programs setup related function*

```
private function setupShaders():void{
        myVertexAssembler.assemble(Cont
        ext3DProgramType.VERTEX,VERTEX_
        SHADER,false);
        myFragmentAssembler.assemble(Con
        text3DProgramType.FRAGMENT,FRAGM
        ENT_SHADER,false);
        myProgram=myC3D.createProgram();
        myProgram.upload(myVertexAssembl
        er.agalcode,myFragmentAssembler.
        agalcode);
        myC3D.setProgram(myProgram);
        screenProjection.perspectiveFiel
        dOfViewRH(camFOV,viewWidth/viewH
        eight,camZNear,camZFar);
        cameraMatrix.appendTranslation(
        0,0,-20);
        objectMatrix.appendTranslation(-
        0.5,-1,-0.5);
        this.stage.addEventListener(Even
        t.ENTER_FRAME,render);
}
```

to the video card, and test it on several different configurations. We are not using stencil or depth buffers in this example, so we set the last parameter to false.

When rendering, it is important to take into account which side of our triangular polygons is visible to the camera. Here, we must know that each visible face is checked against each light on the scene, so if we want to optimize the GPU time, we should use the appropriate culling. In our example, we are using the BACK face setting. It is a closed polyhedral figure, so we will never see the inside looking faces. If the figure was open, as in a box without a lid, then `FRONT_AND_BACK` setting could be appropriate, but would double lighting calculations. Which face is `FRONT` and `BACK` on a triangular polygon is decided by the vertex selection order on the index. That is, a triangle made of vertexes 0, 1 and 2 would be facing opposite to a triangle defined by 0, 2 and 1. Both triangles use the same vertexes, but have a different normal or front facing side.

**Listing 13.** *Render loop with animation handling*

```
        private function render(event:Event):
                void{
                objectMatrix.appendRotation(1,Ve
                ctor3D.Z_AXIS,pivot);
                objectMatrix.appendRotation(2,Ve
                ctor3D.Y_AXIS,pivot);
                objectMatrix.appendRotation(3,Ve
                ctor3D.X_AXIS,pivot);
                transformMatrix.identity();
                transformMatrix.append(objectMa
                trix);
                transformMatrix.append(cameraMa
                trix);
                transformMatrix.append(screenPr
                ojection);
                myC3D.setProgramConstantsFromMat
                rix(Context3DProgramType.VERTEX,
                0,transformMatrix,true);
                myC3D.clear(0.2,0.6,0.8);
                myC3D.drawTriangles(myIBuffer,0,
                indexes.length/3);
                myC3D.present();
        }
```

**Listing 14.** *Error event handler*

```
        private function contextCreationError(err
                or:ErrorEvent):void{
                trace(error.errorID+":
                "+error.text);
        }
}
```

The function ends by calling the polygon and shader setup phases.

The setupPolys function is in charge of filling the VertexBuffer3D object with each vertex position and colour information and the IndexBuffer3D object with the vertexes sets that make up the triangles of our 3D object. The VertexBuffer3D data is used later by the Program3D object as input data to be used by the Vertex and Fragment Shaders. The IndexBuffer3D is used at the render loop as the list of triangles to be rendered.

The setVertexBufferAt method defines the variable pointers that will be used by the shaders programs as parameters. The `FLOAT_3` constant indicates that we will use 3 float values as input parameters. We define the first 3 parameters as x, y and z, and the second 3 parameters as the r, g and b values to be used by the AGAL bytecode (the shaders programs).

The setupShaders function generates the AGAL bytecode using the AGALMiniAssembler object. The AGAL assembler code is different for Vertex Shader and Fragment Shader coding, so it is important to specify the assembler program type as VERTEX or FRAGMENT, as needed. The Program3D object is created and used to upload both Vertex and Fragment Shaders programs into the 3D pipeline.

The second part of the function deals with setting up the 3D matrixes to store camera lenses physical properties, the position of the camera in our virtual world and the object position, as well. We place the camera 20 units backwards from the center of our virtual world, which is at coordinates (0,0,0). That way we are able to see our object. Since we backed from it many units, it will be seen small on our screen. The object is manually placed with its physical center at the center of our 3D virtual world. We achieve this by moving it half its height, width and depth sizes.

We then define an `ENTER_FRAME` listener function which will handle our animation loop.

The render function will be called each and every frame, so we will use it to create our animation loop.

First, we will add up the effect of rotating on each axis our object on its matrix, which will be used as a constants array to pass into the shader program. The rotation requires a pivot point, which is our pivot Vector3D variable defined at the beginning of our program. The transformMatrix object is initialized as an identity matrix (filled with ones at its main diagonal). We then append each the camera and object model matrixes to accumulate all transformations that affect our view of the world. The lens data which defines the perspective attributes is added also.

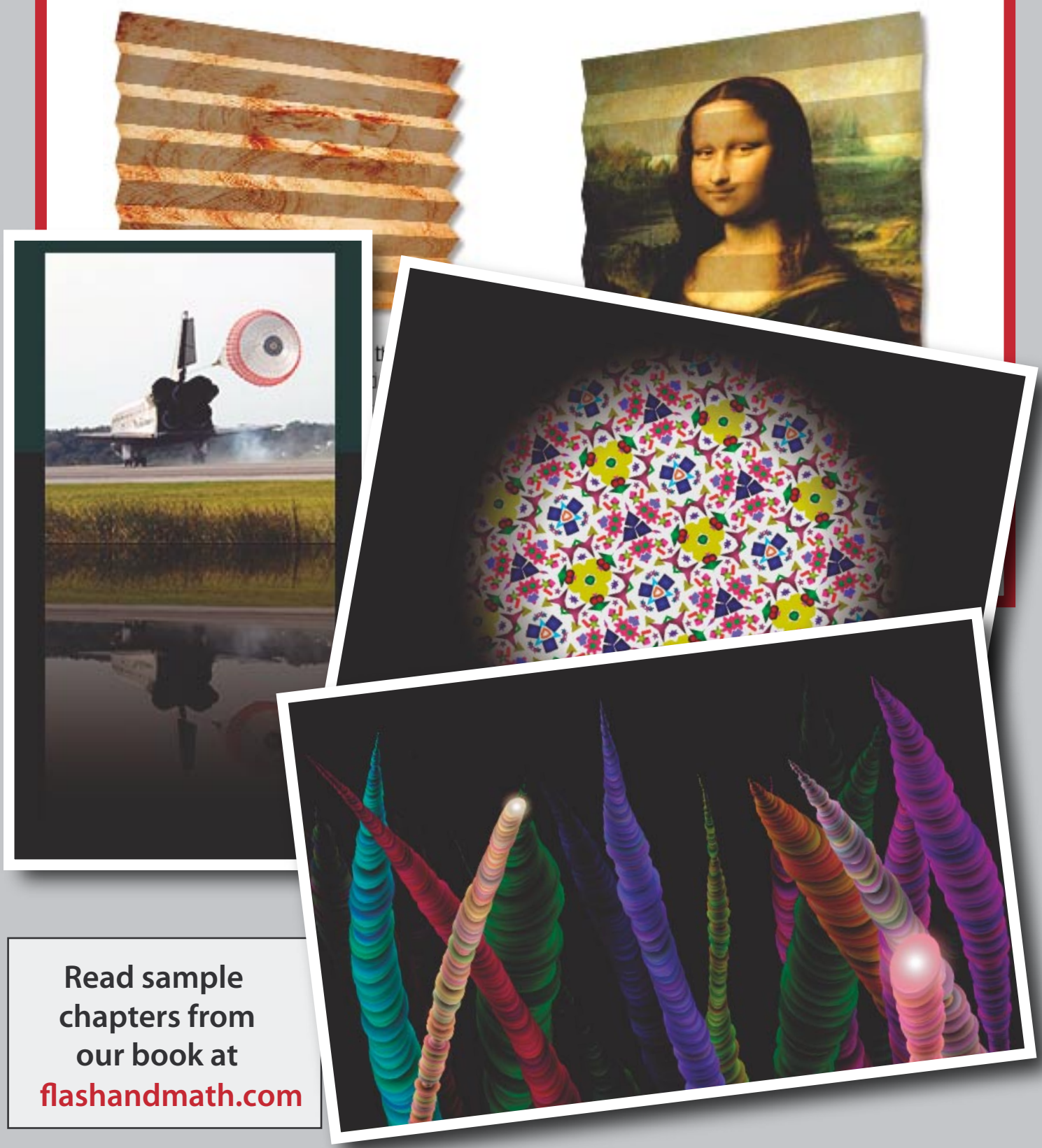
When we have finished with all transformations, we then pass this final matrix as a constant vc0 to our vertex shader program.

Flash & Math

# www.flashandmath.com

ActionScript 3 Tutorials

Dazzling Effects in Flash CS4 and CS5



Read sample chapters from our book at flashandmath.com

**Listing 15a.** *Complete code with comments*

```
package
{
	// These 2 libraries are not official yet
	import com.adobe.utils.AGALMiniAssembler;
	import com.adobe.utils.PerspectiveMatrix3D;

	import flash.display.Sprite;
	import flash.display.Stage3D;
	import flash.display.StageAlign;
	import flash.display.StageScaleMode;
	import flash.display3D.Context3D;
	import flash.display3D.Context3DProgramType;
	import flash.display3D.Context3DRenderMode;
	import flash.display3D.Context3DTriangleFace;
	import flash.display3D.Context3DVertexBufferFor
		mat;
	import flash.display3D.IndexBuffer3D;
	import flash.display3D.Program3D;
	import flash.display3D.VertexBuffer3D;
	import flash.events.ErrorEvent;
	import flash.events.Event;
	import flash.geom.Matrix3D;
	import flash.geom.Vector3D;

	public class S3DTest extends Sprite
	{
		// We define a VGA sized screen for our 3D
		public const viewWidth:Number=640;
		public const viewHeight:Number=480;
		// camZNear defines the depth position of
			our screen
		// camZFar defines how far we are able to
			see
		 public const camZNear:Number=1;
		public const camZFar:Number=500;
		// camFOV defines the focal aperture of
			our camera
		public const camFOV:Number=45*Math.PI/180;

		// We declare our basic Stage3D elements
		private var myS3D:Stage3D;
		private var myC3D:Context3D;
		// These buffers store our list of
			vertexes and triangles
		private var myIBuffer:IndexBuffer3D;
		private var myVBuffer:VertexBuffer3D;

		// Each matrix stores the attributes for
			a different 3D element
		// screenProjection control the camera
			visual attributes related to
			perspective
		// object and camera are described by
			their corresponding Matrix3D
			objects
		// transformMatrix is an auxiliar that
			stores the accumulated result
		//   of all spatial transformations
		private var screenProjection:
			PerspectiveMatrix3D=new
			PerspectiveMatrix3D();
		private var objectMatrix:Matrix3D=new
			Matrix3D();
		private var cameraMatrix:Matrix3D=new
			Matrix3D();
	private var transformMatrix:Matrix3D=new Matrix3D();

		// We need a relative point in space to
			perform rotations of our animation
		private const pivot:Vector3D=new
			Vector3D();

		// The following strings store the AGAL
			programming
		// We need 1 Vertex Shader and 1 Fragment
			Shader to render
		// These are very simple shaders that
			only pass the
		//   position and colour attributes
			values
		private const VERTEX_SHADER:String="m44
			op, va0, vc0 \n"+"mov v0, va1";
		private const FRAGMENT_SHADER:String="mov
			oc, v0";

		// We have 6 data elements for each
			vertex
		// x,y,z and r,g,b values (in that order)
		private const dataPerVertex:int = 6;
		// Describing cuadricular faces, made out
			of 2 triangles each
		private var vertexes:Vector.<Number> =
			Vector.<Number>(
			[
			// x,y,z r,g,b format
			0,0,0, 1,0,0, // front face
			0,2,0, 1,0,0,
			1,2,0, 1,0,0,
			1,0,0, 1,0,0,

			0,0,0, 0,1,0, // bottom face
			1,0,0, 0,1,0,
			1,0,1, 0,1,0,
			0,0,1, 0,1,0,
```

**Listing 15b.** *Complete code with comments*

```
                    0,0,1, 1,0,0, // back face
                    1,0,1, 1,0,0,
                    1,2,1, 1,0,0,
                    0,2,1, 1,0,0,

                    0,2,1, 0,1,0, // top face
                    1,2,1, 0,1,0,
                    1,2,0, 0,1,0,
                    0,2,0, 0,1,0,

                    0,2,1, 0,0,1, // left face
                    0,2,0, 0,0,1,
                    0,0,0, 0,0,1,
                    0,0,1, 0,0,1,

                    1,2,0, 0,0,1, // right face
                    1,2,1, 0,0,1,
                    1,0,1, 0,0,1,
                    1,0,0, 0,0,1
                    ]
                    );
            // Each triangle in our object is made
                    out of 3 vertexes
            // Vertex numbers are their corresponding
                    place in the vertex list
private var indexes:Vector.<uint> = Vector.<uint>( [
                    2,1,0, // front face
                    3,2,0,
                    4,7,5, // bottom face
                    7,6,5,
                    8,11,9, // back face
                    9,11,10,
                    12,15,13, // top face
                    13,15,14,
                    16,19,17, // left face
                    17,19,18,
                    20,23,21, // right face
                    21,23,22
                    ] );

            // Initialize the assembly objects and
                    the GPU program uploader object
            // The Program3D object gets 2 bytecode
                    inputs: 1 vertex shader, 1
                    fragment shader
            private var myVertexAssembler:
                    AGALMiniAssembler = new
                    AGALMiniAssembler();
            private var myFragmentAssembler:
                    AGALMiniAssembler = new
                    AGALMiniAssembler();
private var myProgram:Program3D;

// The start point of our program
public function S3DTest():void
{
        // Standard Stage management in
        Flash
        this.stage.scaleMode =
        StageScaleMode.NO_SCALE;
        this.stage.align = StageAlign.TOP_
        LEFT;
        //this.stage.nativeWindow.activa
        te(); // Use if your output is to
        AIR

        // We work with the lowest Stage3D
        object on the display stack and
        position it
        myS3D = stage.stage3Ds[0];
        myS3D.x = 10;
        myS3D.y = 10;

        // Create 2 event handlers for
        success and error on content
        creation
        // We then request a Context3D
        surface for our Stage3D screen
        myS3D.addEventListener(
        Event.CONTEXT3D_CREATE,
        contextCreated );
        myS3D.addEventListener(
        ErrorEvent.ERROR,
        contextCreationError );
        myS3D.requestContext3D(
        Context3DRenderMode.AUTO );
}

// Beware, Adobe warns that
        context3DCreate event can happen
        at any time,
//   such as when the hardware resources
        are taken by another process
// These function is executed if there
        is no problem with assigning a 3D
        context
private function contextCreated( event:
        Event ):void
{
        // Eet Stage3D context from event
        target
        myC3D = Stage3D( event.target
        ).context3D;
```

**15c.** *Complete code with comments*

```
                // Establish debugging session
                //renderContext.enableErrorChecki
                ng = true; // Can slow rendering,
                use when developing/testing
                // We must set up the GPU render
                surface
myC3D.configureBackBuffer( viewWidth, viewHeight, 2,
                false );
                // We only render back faces.
                Other options are FRONT, NONE and
                FRONT_AND_BACK
                // Use FRONT for an interesting
                optical effect
                myC3D.setCulling( Context3DTriangl
                eFace.BACK );
                // Watch in console if we got the
                right context
                //trace( "3D driver: " +
                myC3D.driverInfo );
                setupPolys();
                setupShaders();
        }

        // This function loads the 3D object
                description into 2 separate
                buffers
        // myIBuffer describes triangles through
                vertex groups
        // myVBuffer describes vertexes position
                and colour
        // myVBuffer will be used by the shader
                programs, and myIBuffer will be
                used as a render list
        private function setupPolys():void
        {
                // These buffers are used as GPU
                program variable inputs
                // myVBuffer stores the variables
                to be used by the vertex shader
                and fragment shader programs
                myVBuffer = myC3D.createVertexBuff
                er( vertexes.length/dataPerVertex,
                dataPerVertex );
                myVBuffer.uploadFromVector(
                vertexes, 0, vertexes.length/
                dataPerVertex );
                // myIBuffer stores the triangle
                sets to be used when rendering
                myIBuffer =
                myC3D.createIndexBuffer(
                indexes.length );
                myIBuffer.uploadFromVector(
                indexes, 0, indexes.length );

                // Assign the vertex variables for
                the vertex program
                // FLOAT_3 means that we pass 3
                float variables
                // We assign the first 3 values to
                x,y,z variables
                myC3D.setVertexBufferAt( 0,
                myVBuffer, 0, Context3DVertexBu
                fferFormat.FLOAT_3 ); // va0 is
                position
                // We assign the second 3 values
                to r,g,b variables
                myC3D.setVertexBufferAt( 1,
                myVBuffer, 3, Context3DVertexBuffe
                rFormat.FLOAT_3 ); // va1 is color
        }

        // This function generates the code for
                the GPU loading
        // It also establishes the initial
                position for each element on our
                scene
        private function setupShaders():void
        {
                // Generate AGAL bytecodes to
                program the GPU
                myVertexAssembler.assemble( Cont
                ext3DProgramType.VERTEX, VERTEX_
                SHADER, false );
                myFragmentAssembler.assemble(
                Context3DProgramType.FRAGMENT,
                FRAGMENT_SHADER, false );

                // We upload the programs into the
                GPU
                // The AGAL code takes care of GPU
                rendering
                myProgram = myC3D.createProgram();
                myProgram.upload( myVertexAssembl
                er.agalcode, myFragmentAssembler.a
                galcode );
myC3D.setProgram( myProgram );

                // Describe the camera physical
                properties with a matrix
                screenProjection.perspectiveFi
                eldOfViewRH( camFOV, viewWidth/
                viewHeight, camZNear, camZFar );
                // Add the camera position
                // We move back ourselves in the
```

Save Hours Building Apps with

# PlugrMan

## The Universal API Toolkit

With support for **AMFPHP**, **Zend AMF** and **SabreAMF**, PlugrMan is the ideal tool for developers who need to build, work with and debug remote AMF services.

- Introspect Remote Services
- Store Test Data
- Automate Service Tests
- Generate Detailed Reports
- Generate ActionScript 3 Code for your Services

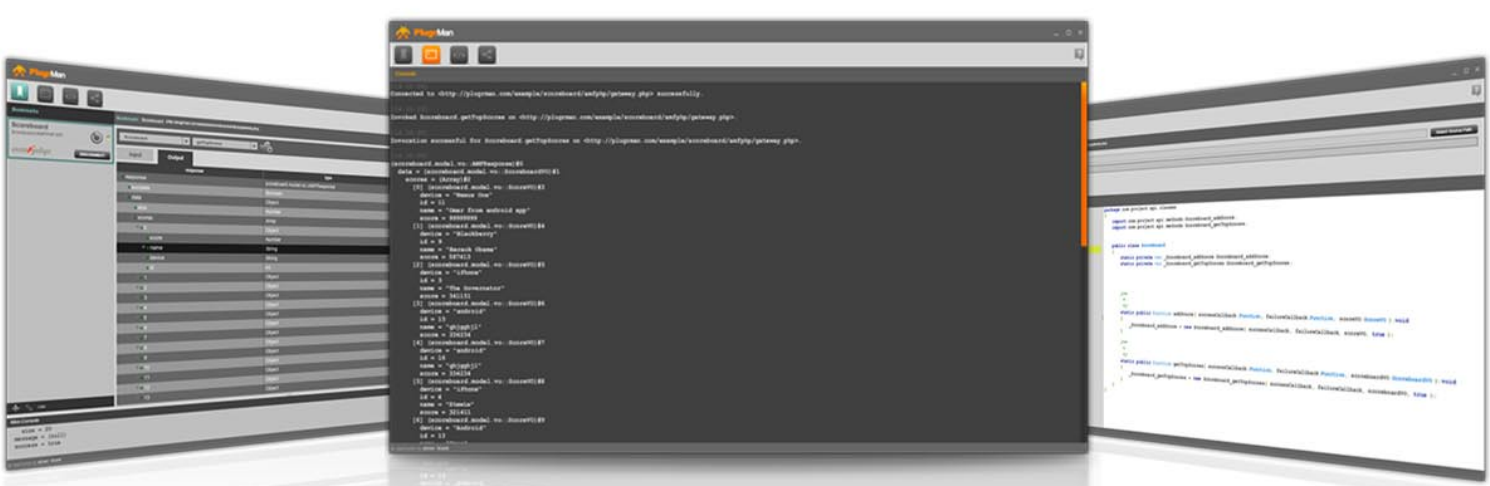
**Only $79.99!**

Get Your **FREE 30-Day Trial** at

plugrman.com

Mac Windows Linux

Runs on **MacOS**, **Windows** & **Linux**!



Introspect and Test Remote Services

Generate Reports on Automated API Tests

Generate AS3 Classes for API Communication

*"It is **totally worth the price**, it will **save you time**."* - GÁBOR CSOMÁK, Flash & Flex Developer's Magazine

almer/blank laboratories

www.plugrman.com | 

**Listing 15d.** *Complete code with comments*

```
            Z axis
            cameraMatrix.appendTranslation( 0,
            0, -20 );    // Move view back
            // Move object into its 3D
            position, and out of the origin
            // We substract half the size of
            the maximum coordinates in each
            axis
            // This way we create a centering
            effect
            objectMatrix.appendTranslation(
            -0.5, -1, -0.5 ); // Center cube
            on origin

            // Animation loop
            this.stage.addEventListener(
            Event.ENTER_FRAME, render );
        }

        // Here we render the scene each and
            every frame
        private function render( event:Event ):void
        {
            // We apply some rotation relative
            to the pivot on each axis
            objectMatrix.appendRotation( 1,
            Vector3D.Z_AXIS, pivot );
            objectMatrix.appendRotation( 2,
            Vector3D.Y_AXIS, pivot );
            objectMatrix.appendRotation( 3,
            Vector3D.X_AXIS, pivot );

            // The transformations are
            cumulatively applied
            // We first generate an identity
            matrix filled with ones on the main
            diagonal
            // It will be used for linear
            transformations
            //   representing transition,
            rotation and scaling
            transformMatrix.identity();
            // Add the object transformation
            data
            transformMatrix.append(
            objectMatrix );
            // Add the camera transformation
            data
            transformMatrix.append(
            cameraMatrix );
            // Add the camera physical
            attributes to render
            transformMatrix.append(
            screenProjection );

            // The transformMatrix is passed
            into the vertex shader as program
            constant, vc0
            // This data is fed into the GPU
            program
            myC3D.setProgramConstantsFromMatri
            x( Context3DProgramType.VERTEX, 0,
            transformMatrix, true );

            // Clear is required before
            drawTriangles on each frame
            // Using a light blue color as
            background in r,g,b
            myC3D.clear( 0.2,0.6,0.8 );

            // Draw the triangles that make up
            each polygon
            // We read the index buffer that
            describe which vertexes are used
            //   to make each triangle
            // We use all available triangles
            on the buffer, from 0 to length/3
            //   because each triangle is made
            out of 3 vertexes
            myC3D.drawTriangles( myIBuffer, 0,
            indexes.length/3 );

            // Everything is now ready to just
            ask the GPU to render our final
            frame
            myC3D.present();
        }

        // There are sometimes when we get a 3D
            context creation error
        // Maybe some other program is accesing
            the GPU
        private function contextCreationError(
            error:ErrorEvent ):void
        {
            trace( error.errorID + ": " +
            error.text );
        }
    }
}
```

The render loop consists of these simple operations: we clear every frame the video buffers, then we draw triangles defined at the IndexBuffer3D object from a range specified by us. In this case, we are rendering from vertex 0 up to the whole length of triangles (we divide by 3 because each triangle is made of 3 vertexes).

And finally, we ask the context to present those triangles. That is, to render them to the screen.

If there is an error on the Context3D creation process, we will catch it and display it on the contextCreationError function.

So that is it. You should have now a red, green and blue colored polyhedral figure if you followed along. And now that you know what is going on each part of it, you should be able to experiment with it. Maybe you could try adding some other viewports to look at the same polygon from several angles as in a classic 3D modeling tool.

I am quite sure that many of you might think that my code is not a very good one, and that it could improve with optimizations, but the idea was not to make a theoretically correct object oriented program, but an easy to follow one.

I think that the AGAL code needs a little more explanation, as well as texturing and some other neat tricks. I will be working on that as well, and the Proscenium and Pixel Bender 3D topics should come also in the future. I also think that it is nicer to simply use a 3D framework to create more advanced programs, but this article was done to show you and explain how things are working at the very low level of AGAL and Stage3D.

In a nutshell, we have covered the importance of these advances on the FP11 and AIR3 Flash platforms. You should now understand the basics of the 3D pipeline that powers it all, and I hope I was able to describe in a clear manner the things that go behind to just render a simple 3D object on screen.

The entire code, with my comments, follows: Listing 15.


## IZCÓATL ARMANDO ESTANOL FUENTES

***Izcóatl Armando Estanol Fuentes is labouring at Universidad La Salle, México as the IT Service Delivery Chief. He is passionate about his family, 3D and Flash animation, film making and technologies applied to education. You can find him around having a good time with his family, researching or making VFX for BYP UK. Contact: iaef@ulsa.mx***