OpenGL®

Pipeline
The Official ARB Newsletter

Issue 001
Q3 2006
Full PDF print edition

In This Issue:

A Message from the ARB Secretary

Welcome to the first issue of OpenGL Pipeline, the official newsletter of the OpenGL Architecture Review Board. Welcome — and goodbye — because this will probably be the *last* issue!

Now that I have your attention: this doesn't mean that the newsletter is going away. But the ARB itself is going away! Why? Where? How? What does this mean for OpenGL standardization? Read on.

When Kurt Akeley and Mark Segal created OpenGL in the early 1990s, the 3D industry was very different. Graphics hardware was restricted to workstations and servers costing tens of thousands of dollars and up. There was no 3D games industry (Id's DOOM wouldn't even come out for a few more years). And hardware was very, very restricted in what it could do.

The ARB was set up to govern OpenGL, drawing on a group of high-end workstation and simulator manufacturers: DEC, Evans and Sutherland, HP, IBM, SGI, and others. But in the late 1990s, graphics hardware started to get cheaper, pervasive, and eventually much more capable, thanks to a new generation of companies like 3dfx, 3Dlabs, ATI, and NVIDIA. The ARB membership has reflected this change. Most of the innovations in OpenGL today come from those "consumer graphics" companies.

Now 3D acceleration is moving to cell phones, and OpenGL is there, too, as OpenGL ES, a subset of OpenGL created in the Khronos Group. Khronos is an entity similar to the ARB, but more widely focused, developing authoring (Collada), digital media/imaging (OpenMAX and OpenML), 3D (OpenGL ES), 2D (OpenVG), and sound (OpenSL ES) APIs.

We've decided that the future health of OpenGL — in all its forms — will be best served by moving OpenGL into Khronos, too. There are many advantages, such as:

The OpenGL and OpenGL ES groups can communicate under the same set of intellectual property rules. IP rules are to standards like dental checkups are to you: unpleasant, but essential to avoid pain in the future.

- OpenGL and OpenGL ES might converge back into a single API. Mobile devices have grown more powerful and added back many features missing from OpenGL ES 1.0. And with programmable graphics pipelines common, we may be ready to phase out redundant and legacy features from OpenGL.

- The OpenGL ARB Working Group can work closely with other APIs in Khronos. For example, we might eventually replace the GLX/WGL/AGL APIs with EGL, a cross-platform equivalent developed in Khronos.

- The OpenGL ARB Working Group and the rest of Khronos can pool efforts on SDKs and documentation. For example, the OpenGL extension registry will grow into a registry for all the Khronos APIs.

- Finally, OpenGL and Khronos can more efficiently share administrative, logistical, and website support from the Gold Standard Group.

From a developer's viewpoint, there'll be little change. The opengl.org website and boards will continue, though we may merge the underlying webhost with khronos.org. The standards process will operate much as it does today, although we will coordinate our releases and announcements with other Khronos APIs.

Not much will change in our day-to-day operation, either. Khronos and ARB processes are very similar. Other Khronos member companies will be able to join in our working groups.

Merging is a complicated process and will take months to complete, but is well underway. So, the next quarterly issue of "OpenGL Pipeline" will probably be published by the Khronos Group, not the ARB, and will probably be expanded to cover OpenGL ES and perhaps other Khronos APIs. We'll talk more about the status of the merger at the SIGGRAPH OpenGL BOF Session.

It's been my privilege and my pleasure to serve as the OpenGL ARB Secretary since joining SGI in 1997. Now I'm looking forward to a new stage in the evolution of OpenGL. Come along for the ride!

Jon Leech
ARB Secretary

OpenGL PIPELINE
THE OFFICIAL ARB NEWSLETTER

A Welcome Message from the Ecosystem Working Group

BENJ LIPCHAK
ATI, Ecosystem Working Group Chair

"Rumors of my demise have been greatly exaggerated."
— Mark Twain

Welcome to the first edition of OpenGL Pipeline, the quarterly newsletter covering all things the OpenGL standards body has "in the pipeline." Each issue will feature status updates from the various active working groups, along with a handful of thoughtful articles, event announcements, and product spotlights. Then if there's any room leftover, we'll throw in a semi-informative rambling or two. All we can promise is that this publication will be worth every penny you've paid for it.

The Ecosystem Working Group was formed in March of this year. Its charter is to tackle "everything else." We leave the heavy lifting — debating new OpenGL features, generating new APIs, and writing extension specs — to the other highly skilled working groups. They are the rock stars. In contrast, the Ecosystem WG is the unsung hero working backstage to increase the impact of all those new features. We are the wind beneath their wings, if you will.

According to the American Heritage Dictionary, the word ecosystem means "an ecological community together with its environment, functioning as a unit." To us in the Ecosystem WG it means all of the resources on the periphery serving as a development environment to make OpenGL more useful or accessible to you, the community. We started by conducting a poll on opengl.org to find out what you were most interested in our tackling first. Was it reference materials? Tutorials & sample code? Tools & utilities? A test suite? No. 67% of you chose "OpenGL SDK: a single SDK sponsored by the ARB, endorsed by all vendors, with some/all of the above." In other words, you want it all. We get it.

Ecosystem WG activities over the last quarter have included the following: planning for the launch of an OpenGL SDK, establishing a modern toolchain for generating reference documentation, and revamping naming convention guidelines for the other working groups to utilize when creating future APIs. Work next quarter will focus on generating OpenGL 2.1 reference materials and starting to piece together the SDK, soliciting contributions from the OpenGL community at large.

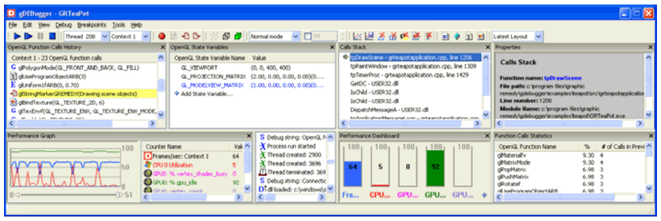
The second most popular response to the poll was "Better communications: what has the ARB been doing and what are its future plans?" which segues nicely into this newsletter. Regardless of the poll, you may find yourself asking, "Why, after all these years, is the OpenGL standards body finally opening up and sharing with its audience, its devoted developers, its enthusiastic end-users, its *people*?" It must be a maturity thing. It took us a solid 14+ years to shed our youthful shyness and find a voice. The last decade was just an awkward phase. We're over it now. This newsletter is just the beginning of OpenGL's long anticipated coming of age.

"I'm not dead yet!" — Monty Python

Free OpenGL Debug Tools for Academic Users

Here is some great news for students and academic OpenGL users! The OpenGL ARB and Graphic Remedy have crafted an Academic Program to make the full featured gDEBugger OpenGL debug toolkit available for use in your daily work and research — free of charge!

gDEBugger, for those of you who are not familiar with it yet, is a powerful OpenGL and OpenGL ES debugger and profiler delivering one of the most intuitive OpenGL development toolkits available for graphics application developers. gDEBugger helps you save precious debugging time and boost your application's performance. It traces application activity on top of the OpenGL API to provide the necessary information to find bugs and to optimize application rendering performance.



The ARB—Graphic Remedy Academic Program will run for one year during which time any OpenGL developer who is able to confirm they are in academia will receive an Academic gDEBugger License from Graphic Remedy at no cost. This license will be valid for one year and will include all gDEBugger software updates as they become available. Academic licensees may also optionally decide to purchase an annual support contract for the software at the reduced rate of $45 (or $950 for an academic institution).

There are also a limited number of free licenses available for non-commercial developers who are not in academia.

gDEBugger is rapidly developing a strong following. It is already being used in many universities and by graphics hardware vendors such as NVIDIA and ATI. It is being put to use in the realms of game development, film, visual simulations, medical applications, military and defense applications, CAD, and several other markets. There is no need to make any changes to your source code or recompile your application. Simply run your application in gDEBugger and start tuning it. gDEBugger works with all current graphic hardware products. It supports NVIDIA GPU performance counters via NVPerfKit, NVIDIA GLExpert driver reports, ATI GPU Performance Metrics, the latest version of OpenGL and many additional OpenGL and WGL extensions. It is available for the Windows operating system with a Linux version under development. The Windows and future Linux versions are part of the ARB—Graphic Remedy Academic Program. gDEBugger ES, which supports OpenGL ES, is available for purchase separately.

Graphic Remedy, the makers of gDEBugger, specializes in software applications for the 3D graphics market, specifically tools for 3D graphics developers. The company's mission is to design innovative tools that make 3D graphics programming faster and easier, to save programmers time and money, and to improve graphics application performance and reliability. The company is a Contributor member in the OpenGL ARB and in the Khronos Group.

For further information, visit:
http://academic.gremedy.com/?pipeline

Avi Shapira
Graphic Remedy

Superbuffers Working Group Update

As you might have heard, the scope of the Superbuffers Working Group has broadened considerably. After we finished the EXT_framebuffer_object extension, which you all know and love (I hope!), we started working on adding some missing functionality. Some of you expressed interest in features like rendering to one and two component render targets, as well as being able to mix attachments of different dimensions and even different formats. But most importantly, you wanted to be able to find out how to set up a framebuffer object that is guaranteed to be supported by the OpenGL implementation your application is running on. In other words, how can you set up a framebuffer object so that the dreaded `GL_FRAMEBUFFER_UNSUPPORTED` error will not occur? We worked on a solution for this, but started to realize that this was really hard due to some choices we made in EXT_framebuffer_object. Looking ever deeper, we realized that the current object model in OpenGL is, in large part, to blame for this. As a result, we are now working on a new object model for OpenGL. You might have seen the presentation at GDC. A summary is described here:

http://www.gamedev.net/columns/events/gdc2006/article.asp?id=233

The goals of the new object model are several. First, we want to provide top rendering performance. The current object model has performance cost associated with a name lookup every time an object name is passed to the OpenGL driver. This cost is only going to increase due to the widening gap between CPU and GPU performance. Second, there is a performance cost every time you make a draw call (`glBegin`, `glDrawArrays`, etc.). The OpenGL driver needs to perform a non-trivial amount of validation work before starting to draw. This is especially important if the draw call only consists of a few primitives. Third, we want to eliminate difficult race conditions which arise when sharing objects across multiple OpenGL contexts. For example, what happens when you change the filter mode of a texture object in one context while also using that texture object in another context? Last, but not least, we want to make the new object model simpler to use. State-based errors are a pain to deal with. Say, for example, that one part of your code calls `glActiveTexture`, another part of your code binds a texture object, and a third part of your code sets the filter mode for that texture object — at least, you hope. The active texture state might not be what you wanted it to be at that time. We're going to change this model of binding objects just to set a parameter. In the new object model, any command that sets a parameter of an object will take a handle to that object. No more confusion! Furthermore, object creation will, if successful, always return a handle to the newly created object. The application can no longer make up a name for an object. This is a key component of the new object model, and will help us achieve the goals just outlined.

We will be posting updates to **www.opengl.org** whenever we have something to share. Watch that space!

Barthold Lichtenbelt
NVIDIA, Superbuffers Working Group Chair

New Texture Functions with Awkward Names to Avoid Ugly Artifacts

One problem with many of the extension specs that we face is that they are too often short on motivating examples. Even when there are examples, they suffer from dreaded ASCII art. With this newsletter, I can not only put in a few more examples, I can replace the dreaded ASCII art with the less dreaded programmer art.

Note, these example procedural shaders *will* alias. And since they will alias, why not use an aliased source texture as well?

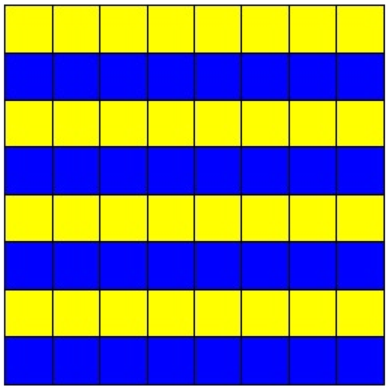


Figure 1 – Source texture – aliasing yellow and blue stripes!

So let's start with a trivial shader: apply this texture to a quad. The quad has texture coordinates `myTC` that are passed in from the vertex shader. `myTC` coordinates are 0.0, 0.0 at the lower left corner and 1.0, 1.0 at the upper right corner.

```
// Fragment Shader 1 - simple texture
varying vec2 myTC;
uniform sampler2D myStripeMap;
```

```
void main(void)
{
    gl_FragData[0] = texture2D(myStripeMap, myTC);
}
```

The technical director asks for a shader that replaces the left side of the texture with lime green. You write the shader (knowing better than to ask why) and add a new control, `mySlider`. When `myTC.s` is less than `mySlider`, the color is green. When `myTC.s` is greater than or equal to `mySlider`, the color is the source texture.

```
// Fragment Shader 2 - left green/right textured
varying vec2 myTC;
uniform sampler2D myStripeMap;
uniform float mySlider;

const vec4 green = vec4( 0.0, 1.0, 0.0, 1.0 );
void main(void)
{
    if (myTC.s < mySlider)
        gl_FragData[0] = green;
    else
        gl_FragData[0] = texture2D(myStripeMap, myTC);
}
```

A quick check with `mySlider` set to 0.5 and everything looks great!

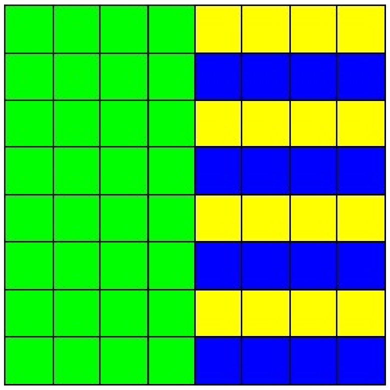


Figure 2 – The textured quad when `mySlider` = 0.5

You are about to ship the shader off to the technical director, but you try a value a bit larger than 0.5. Where did the vertical gray stripe come from?

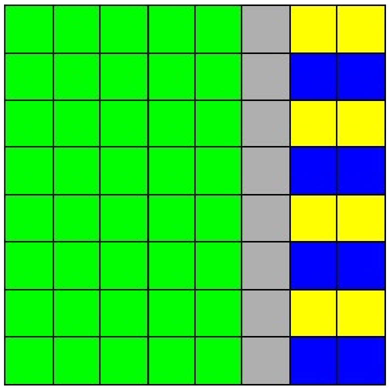


Figure 3 – Sometimes there's a vertical gray strip.

The problem is that the texture fetch is *inside* varying control flow. A mipmapped texture fetch or an anisotropic fetch will calculate an implicit derivative for lambda or the line of anisotropy. Derivatives (explicit or implicit) inside of varying control flow are undefined! Your graphics card happens to either get an answer that sometimes seems right when the texels are far from the conditional. But it also seems to get them very wrong near the conditional, and you guess that the derivatives are very very large near the conditional. The large derivatives near the conditional drive the texture fetches to the *bottom* of your mipmap pyramid. That's why you see the gray vertical stripe.

Note that undefined derivatives mean that different implementations can get very different answers. In fact, you test your shader on an older system and find out that the older system happens to *always* give you the "right" answer!

You rewrite the shader to move the texture fetch outside of control flow and wish there was a better way.

```
// Fragment Shader 3 - with old texture functions
varying vec2 myTC;
uniform sampler2D myStripeMap;
uniform float mySlider;

const vec4 green = vec4(0.0, 1.0, 0.0, 1.0);
void main(void)
{
    vec4 texel = texture2D(myStripeMap, myTC);
    if (myTC.s < mySlider)
        gl_FragData[0] = green;
    else
        gl_FragData[0] = texel;
}
```

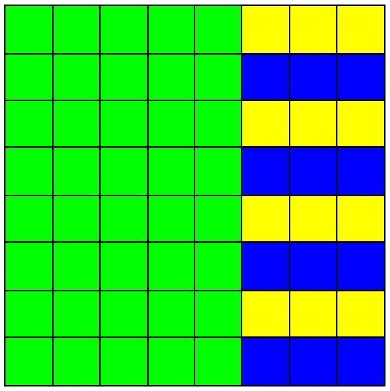


Figure 4 – the correct picture

With a new extension under development, you have another choice besides moving the texture fetch outside of control flow.

An extension proposed in the GLSL Working Group, ARB_shader_texture_lod, adds new built-in texture functions that allow the shader writer to explicitly supply the derivatives. You can calculate the derivatives outside of control flow and fetch the texel inside of control flow.

```
// Fragment Shader 4 - with new texture functions
// NOT YET APPROVED AT THE TIME OF THIS WRITING!
#extension ARB_shader_texture_lod require
```

```
varying vec2 myTC;
uniform sampler2D myStripeMap;
uniform float mySlide;

const vec4 green = vec4(0.0, 1.0, 0.0, 1.0);
void main(void)
{
    vec2 dPdx = dFdx(myTC);
    vec2 dPdy = dFdy(myTC);
    if (myTC.s < mySlide)
        gl_FragData[0] = green;
    else
        gl_FragData[0] =
            texture2DGradARB(myStripeMap, myTC,
                             dPdx, dPdy);
}
```

Shader 3 and Shader 4 will both get the correct answers on all implementations, but the latter may be more efficient on some implementations.

In summary, existing texture functions may need to calculate implicit derivatives for mipmapped texture fetches or anisotropic texture fetches. Derivatives inside of varying control flow are undefined. New texture functions are introduced by ARB_shader_texture_lod with explicit derivative parameters. This allows a shader writer to move the derivatives outside of varying control flow while keeping the texture fetch inside of control flow.

Bill Licea-Kane
ATI, GLSL Working Group Chair

Improved Synchronization Across GPUs and CPUs — No More glFinish!

The Async Working Group recently finished the ARB_sync specification. It provides a synchronization model that enables a CPU to synchronize with a GPU OpenGL command stream across multiple OpenGL contexts and multiple CPU threads. This extension, for example, allows you to find out if OpenGL has finished processing a GL command without calling `glFinish`. As you know, `glFinish` is a heavyweight operation that you really should not call more than once per frame. Calling it is so expensive because it drains all commands that OpenGL has buffered up before resuming processing.

This extension also allows you, for example, to synchronize rendering in one context with rendering in another context without calling `glFinish`. Say you are rendering to a texture in one context while another context needs to use the result of that rendering. You do this by inserting a fence right after the rendering to texture commands in the one context and waiting for the fence to complete in the other context. Again, there is no need to call `glFinish`!

A link to the actual extension and a discussion are here: **http://www.opengl.org/discussion_boards/ubb/ultimatebb.php?ubb=get_topic;f=3;t=014377**. Please let us know how you would use this extension, what you think is good about it, and what needs some work.

Currently the Async Working Group is transforming this extension into the new object model that the superbuffers Working Group is working on. We are also starting to look at extending the ARB_sync extension to provide synchronization with, for example, each vertical retrace (vblank) and adding the capability to figure out at what time exactly a fence completed. Another topic on our agenda is to look at so-called 'predicated rendering.' Think of this as an occlusion query test, where the result of the test automatically controls whether a set of geometry is rendered by the GPU, without any CPU intervention.

Barthold Lichtenbelt
NVIDIA, Async Working Group Chair

OpenGL ARB & Khronos Group SIGGRAPH 2006 Schedule

All Events take place at the Boston Convention and Exhibit Center. Come visit Khronos at Booth #611 on the main floor!

Khronos & OpenGL ES Tech Talks (DevU)	
Room:	Room 206A
Date:	**Wednesday, 2 August**
Time:	10am - 3:30pm

OpenGL BOF	
Room:	Room 206A
Date:	**Wednesday, 2 August**
Time:	4pm - 6pm

COLLADA BOF & Social Event	
Room:	Room 206A
Date:	**Wednesday, 2 August**
Time:	6pm - 8pm

OpenGL ES BOF	
Room:	Room 251
Date:	**Thursday, 3 August**
Time:	10am - 12 noon

COLLADA Tech Talk	
Room:	Room 251
Date:	**Thursday, 3 August**
Time:	12 noon - 2pm: (4 x 30 minute presentations from SCE, NVIDIA, Feeling, Softimage)