Introduction

In modern workstation and gaming systems it is becoming increasingly more common to see multiple GPUs. Performance gains can be seen on these systems by enabling ATI CrossFire™ mode. Multiple GPUs also enable a system to support more display devices. There are many benefits to having multiple graphics cards in a workstation.

The computing industry has seen similar trends with CPUs. Originally multiple CPU configurations were reserved for servers and provided little benefit on end user PCs. In the last few years, systems with at least 2 CPUs have become very common. More importantly, we now have applications and OSes that can take better advantage of the multiple processing units.

The industry is again on the verge of a shift as graphics processors have become smaller, more powerful and have found their way into commonly used devices. What is currently lacking is a way to make efficient use of multiple GPUs, especially if there are more than two, or when not configured for ATI CrossFire.

GPU Association – Picking a GPU

AMD has provided a method for addressing the problem of not having access to all GPUs, specifically for OpenGL. A new extension, WGL_AMD_GPU_association, allows applications access to specific AMD GPUs. This allows applications to make intelligent decisions about how to most efficiently allocate and execute rendering tasks.

This extension is available on Catalyst 9.3 and newer driver kits. The extension is written against OpenGL 1.5, but is really a WGL extension that has limited direct interaction with GL. It will also function equally well on OpenGL 2.0, 2.1, 3.0, 3.1, and beyond. Currently only version OpenGL 2.1 and later are available on AMD drivers, OpenGL 2.1 is backwards compatible with OpenGL 1.5.s.

WGL_AMD_GPU_association provides several key pieces of functionality that make efficient distribution of rendering possible. The first is a method of determining what graphics resources are available in a system. Next is a way to allocate OpenGL contexts on specific GPUs and query where non-associated contexts have been allocated. Last is a new path for fast and efficient data transfer between contexts.

Determining Topology

GPU Count

In order to make efficient use of a system’s graphic resources, an application must first know what GPUs are available. WGL_AMD_GPU_association has provided a set of interfaces to do exactly that. wglGetGPUIDsAMD allows an application to get an enumerated list of all graphics processors in the system. These are each identified by a unique ID. IDs are valid for the current instance of the operating system and current configuration. Rebooting a system, disabling resources, or otherwise fundamentally
changing the system configuration could invalidate the list of IDs. Caching the IDs between reboots and expecting all resources to be the same is invalid.

```c
UINT wglGetGPUIDsAMD(UINT maxCount, UINT *ids);
```

To use this function, allocate an array of unsigned integers and pass a pointer this into `wglGetGPUIDsAMD` along with the size of the array, an array sized at 16 values will be sufficient for the near future. The value returned by this function is the actual number of GPUs available in the system. In most cases, this value will be the number of IDs written to the `ids` array. However, if the array size was too small to handle the complete list, the returned number available GPUs will be larger than `maxCount` passed into the function. In this case only `maxCount` values will be written to the array.

The ID 0 will never be returned. If the `ids` pointer was null, no values will be written, but the total number of GPUs in the system will still be returned. An application can first call `wglGetGPUIDsAMD` to determine an appropriate size for the array.

**GPU Properties**

Once the list of GPUs is known, an application will need to determine what each GPU is capable of in order to make an educated decision on how to distribute rendering. The function `wglGetGPUInfoAMD` can be used to find out what each GPU is capable of.

```c
INT wglGetGPUInfoAMD(UINT id, WGL_ENUM property, ENUM dataType, UINT size, void *data);
```

To determine the capabilities of a GPU enumerated in `wglGetGPUIDsAMD`, pass this ID into the `id` field of `wglGetGPUInfoAMD`. The properties that can be queried are listed in Table 1 below. These enums are passed into the `property` field. The `dataType` field is used to specify the return type of the data requested by the application. Valid values are `GL_UNSIGNED_INT`, `GL_INT`, `GL_FLOAT`, and `GL_UNSIGNED_SHORT`. If the `dataType` value is not an appropriate match for the `property` requested, the function will simply return -1. For instance, applications should not use type `GL_FLOAT` to query string values.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WGL_GPU_OPENGL_VERSION_STRING</td>
<td>Returns the OpenGL version for this GPU. This corresponds to a call to <code>glGetString(GL_VERSION)</code>, but can be done before creating an OpenGL context. Data type <code>GL_UNSIGNED_SHORT</code> should be used, and the values will be returned in an array with the array size returned by the function.</td>
</tr>
<tr>
<td>WGL_GPU_RENDERER_STRING</td>
<td>Returns the renderer string for this GPU. This corresponds to a call to <code>glGetString(GL_RENDERER)</code>, but can be done before</td>
</tr>
</tbody>
</table>
creating an OpenGL context. This will be the proprietary GPU name. Data type `GL_UNSIGNED_SHORT` should be used, and the values will be returned in an array with the array size returned by the function.

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WGL_GPU_FASTEST_TARGET_GPUS</td>
<td>Returns an array of GPU IDs ordered from the fastest at index 0 to the slowest at index size-1. The method to determine GPU ordering is proprietary, but will include GPU family as well as clock speeds. This is not simply a list sorted by clock speed.</td>
</tr>
<tr>
<td>WGL_GPU_RAM</td>
<td>Returns the amount of GPU RAM in megabytes. This is a single value.</td>
</tr>
<tr>
<td>WGL_GPU_CLOCK</td>
<td>Returns the GPU clock frequency in megahertz. This is a single value.</td>
</tr>
<tr>
<td>WGL_GPU_NUM_PIPES</td>
<td>Returns the number of 3D pipes on the GPU. This is a single value.</td>
</tr>
<tr>
<td>WGL_GPU_NUM_SIMD</td>
<td>Returns the number of SIMD units in each shader pipe. This is a single value.</td>
</tr>
<tr>
<td>WGL_GPU_NUM_RB</td>
<td>Returns the number of render backends. This is a single value.</td>
</tr>
<tr>
<td>WGL_GPU_NUM_SPI</td>
<td>Returns the number of shader parameter interpolators. This is a single value.</td>
</tr>
</tbody>
</table>

Table 1 – Property values accepted by wglGetGPUInfoAMD

These GPU properties can be queried and used to find the best GPU match for a specific rendering task. For instance, one context may be heavily texture or renderbuffer dependent and the application should use `WGL_GPU_RAM_AMD` as the first sorting parameter. A second context may have extensive shader computations, for which the application could use `WGL_GPU_NUM_SIMD` to determine the best GPU.

Creating Contexts

On AMD hardware, an OpenGL context will automatically be associated with the card attached to the display the window was created on. For example, if an application creates a window and an OpenGL context on display 2 which is attached to a secondary card, the OpenGL context will run natively on this secondary card.

After an application has determined which contexts to use based on the resources and capabilities of each GPU, it can create contexts using appropriate IDs. Applications may always want to create a local, native context on the primary card. Once this is done, information regarding the card this unassociated context was created on can still be queried by getting the ID using `wglGetContextGPUIDAMD` and
then performing the queries as described above. This will provide the information necessary to make sure that off-screen contexts do not collide with the native context GPU.

```c
UINT wglGetContextGUIDAMD(HGLRC hglrc);
```

To find the ID value for a previously created context, use the `wglGetContextGUIDAMD` function. The HGLRC passed into this function can be from an associated context, or from a generic context created by calling `wglCreateContext` or `wglCreateContextAttribsARB`. The value returned will be the ID of the GPU the context is tied to. If the HGLRC passed in is invalid or if an error has occurred, the function will fail and return 0.

Creating an associated context can be done by simply calling `wglCreateAssociatedContextAMD`. This function behaves similarly to `wglCreateContext` but takes a GPU ID instead of a hDC.

```c
HGLRC wglCreateAssociatedContextAMD(UINT id);
```

Use the GPU ID for the GPU this context is intended to run on. A device handle is not necessary because this context will not be attached to a display device. Instead it is attached to the specified GPU. It is important to note that this context will not be associated with or attached to a window.

Additionally, a specific type of associated GL context can be created by using the `wglCreateAssociatedContextAttribsAMD` version. The attributes specified here are the same as those specified for `wglCreateContextAttribsARB`. This function allows applications to specify the version and type of associated context to be created.

```c
HGLRC wglCreateAssociatedContextAttribsAMD(UINT id, HGLRC hShareContext, const int *attribList);
```

To delete an associated context, call `wglDeleteAssociatedContextAMD`. This function will only take HGLRCs that were created by calling `wglCreateAssociatedContextAMD`. If a HGLRC was not created with `wglCreateAssociatedContextAMD`, the function will fail and return false. Note that associated contexts should not be deleted by calling `wglDeleteContext`. This call will also fail and may result in undefined behavior.

```c
BOOL wglDeleteAssociatedContextAMD(HGLRC hglrc);
```

### Rendering with an Associated Context

Once associated contexts are created, they can be bound for use by calling `wglMakeAssociatedContextCurrentAMD`. The same rules apply to this function as to `wglDeleteAssociatedContextAMD`. HGLRCs must be created by calling `wglCreateAssociatedContextAMD` or `wglCreateAssociatedContextAttribsAMD`, and associated contexts must not be deleted by calling `wglDeleteAssociatedContextAMD`.

```c
BOOL wglMakeAssociatedContextCurrentAMD(HGLRC hglrc);
```
Note that only one context can be current to a thread at a time, regardless of which creation function was used to generate them.

A method to query the currently bound associated context is also provided through wglGetCurrentAssociatedContextAMD. Call this function to get the current associated context. If none is current, the function will return NULL. If a non-associated context is current, the function will also return NULL.

\[
\text{HGLRC wglGetCurrentAssociatedContextAMD(void);}\]

After making an associated context current an application will have to do several things before the context can be used. Because there are no windows attached to the associated context, there are also no drawable surfaces (or readable). Essentially, the default framebuffer object is invalid for rendering. An error will be thrown if attempted. The reasoning behind this is that the GL does not know what rendering the associated context will be used for. Applications have complete flexibility to create whatever surfaces they desire. Additionally, the associated context will not use GPU resources by allocating a drawable surface that most applications will not use.

To start, create renderbuffers with the desired size and format. Then attach them to a framebuffer object attach that object. Once the new framebuffer is FRAMEBUFFER_COMPLETE, the context can be used for rendering. Use the context as any GL context with a framebuffer object attached.

Use of wglSwapBuffers or wglSwapLayerBuffers has no effect on an associated context. There is no window attached to the context.

**Sharing Pixel Data Between Contexts**

One way to move data between contexts is to copy pixels or data out of GPU memory and into system memory. Then the data can be copied back up to the other context. This is functional, but not very efficient. Another option is to share data between contexts and access the data directly. However associated contexts that are not associated with the same GPU cannot share data because they do not reside on the same physical hardware.

A new data transfer function has been created to allow an application to transfer data between contexts quickly and efficiently. This function is called wglBlitContextFramebufferAMD. It can be used to push data from the attached framebuffer in the current context to another context. This interface follows all of the rules defined in EXT_framebuffer_blit for the glBlitFramebufferEXT interface.

\[
\text{VOID wglBlitContextFramebufferAMD(HGLRC dstCtx, GLint srcX0, GLint srcY0, GLint srcX1, GLint srcY1, GLint dstX0, GLint dstY0, GLint dstX1, GLint dstY1, GLbitfield mask, GLenum filter);}\]
This provides a mechanism for applications to allow the OpenGL driver to do the data copy instead. The driver is aware of the fastest mechanism for transferring data between contexts, which may take several paths that are considerably faster than copying to system memory and over to the second GPU.

Specify the destination context using the dstCtx parameter. The source and destination regions are specified through the src and dest parameters. The mask parameter allows for selection of specific renderbuffer types; color, depth and/or stencil. The filter parameter specifies how the source image is interpolated when the stretching is necessary.

All error behavior specified for EXT_framebuffer_blit is also applicable to WGL_AMD_GPU_association. Additionally, the source context (current context) cannot be used as the destination context. The destination context must be a valid context. A context must be current at the time of this call. All of these conditions will generate GL_INVALID_OPERATION errors in the GL error stream. Make sure the proper framebuffers are bound to the GL_DRAW_FRAMEBUFFER_EXT and GL_READ_FRAMEBUFFER_EXT attachments in the destination and source contexts.

wglBlitContextFramebufferAMD does not perform any synchronization on attached surfaces. The application must ensure all pending rendering operations are complete on both the source and destination surfaces before executing the blit call.

**Alternatives**

There are several existing methods for distributing rendering, although each has limitations. First, an application can manually create contexts and windows on multiple GPUs. Data copying between contexts would have to be done through the CPU. The window drawables for the additional contexts will likely be wasted as the application would in most cases not want to display off-screen rendering. One of the bigger limitations to this approach is that the application does not know the capabilities of the GPUs it is executing on. The application also cannot be certain that these contexts are actually executing on separate GPUs.

Another alternative is to use WGL_NV_GPU_affinity. This extension also allows for selecting a target GPU for a context. It uses a DC tied to a GPU to accomplish this. This method can be error prone because of the requirement to match affinity DCs with affinity contexts. It also requires setting a pixel format for a specific DC which is then inherited by contexts, even though rendering will generally be off-screen. The WGL_NV_GPU_affinity extension also does not provide a method for efficiently copying data between contexts.

**Using WGL_AMD_GPU_association**

Before using WGL_AMD_GPU_association, an application should test for its existence.

```c
const char * extensions = wglGetExtensionsStringARB(g_hDC);
if (strstr(extensions, "WGL_AMD_gpu_association") != NULL)
{
    // Get WGL_AMD_GPU_association entrypoints
```
The main rendering context should be setup as always. This context will be used for displaying the output directly to the application window.

```cpp
g_hRCMain = wglCreateContext( g_hDC );
wglMakeCurrent( g_hDC, g_hRCMain );
```

Also determine the GPU name for the main context.

```cpp
int nMainGPUID = wglGetContextGPUIDAMD(g_hRCMain)
```

Next the list of GPUs available for associated rendering can be queried. Note that one of the GPUs will be `nMainGPUID` which was returned in the above call and will also be used for displaying to the window.

```cpp
UINT uiGPUIDs[16] = { 0 };
UINT maxGPUs = wglGetGPUIDsAMD(16, uiGPUIDs);

if (maxGPUs == 16)
{
    // The size of uiGPUIDs may not have been large enough to
    // hold all GPUs available, call again with a larger array
}
```

Once the GPU names are known, the individual attributes can be determined. For simplicity, we will only compare GPU clock speed to prioritize GPU usage. But applications should take all relevant information into account when choosing which GPU to use for a particular rendering task.

```cpp
int nReturnedDataCount = wglGetGPUInfoAMD(uiGPUIDs[i], WGL_GPU_CLOCK_AMD, GL_UNSIGNED_INT, 16, intData[i]);
if (nReturnedDataCount != 1)
{
    // wglGetGPUInfoAMD failed. Possibly invalid GPU name used.
}
```

Now that we have the clock speeds for the GPUs available, find the best candidate for off-screen processing.

```cpp
// find the fastest GPU that is not used for displaying to the window
int nFastestGPU = -1;
int nFastestGPUSpeed = -1;
for (int j = 0; j < maxGPUs; j++)
{
    if ((uiGPUIDs[j] != nMainGPUID) &&
        (intData[j] > nFastestGPUSpeed))
    {
        nFastestGPU = j;
        nFastestGPUSpeed = intData[j];
    }
}
nFastestGPU = uiGPUIDs[j];
nFastestGPUSpeed = intData[j];
}

Check the highest supported OpenGL version for the fastest GPU.
char charData[64] = { 0 };
nReturnedDataCount = wglGetGPUInfoAMD(nGPUID,
WGL_GPU_OPENGL_VERSION_STRING_AMD,
GL_UNSIGNED_BYTE, 64, charData);
if(nReturnedDataCount < 1)
{
    // An error occurred
} else if ((charData[0] == '3' &&
    charData[2] >= '1') ||
    (charData[0] >= '3'))
{
    // Can support an OpenGL 3.1 or greater context
}

Create a new associated context for off-screen rendering on the candidate we just selected. Specify a specific context version.
int attribList[5] = {
    WGL_CONTEXT_MAJOR_VERSION_ARB, 3,
    WGL_CONTEXT_MINOR_VERSION_ARB, 2,
    NULL
};
HGLRC hOffScrCtx = wglCreateAssociatedContextAttribsAMD(uiGPUIDs[i],
    NULL, attribList);
if (hOffScrCtx == 0)
{
    // An error occurred
}

Now that we have an associated context for off-screen rendering, make it current and render to it.
wglMakeAssociatedContextCurrentAMD(hOffScrCtx);

// Setup render target
UINT nShadowPassFBOName = 0;
glGenFramebuffers(1, &nShadowPassFBOName);
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, nShadowPassFBOName);

UINT nShadowPassRBOName = 0;
glGenRenderbuffers(1, &nShadowPassRBOName);
glBindRenderbuffer(GL_RENDERBUFFER, nShadowPassRBOName);
glRenderbufferStorage(GL_RENDERBUFFER, 1, DEPTH_COMPONENT24, 1024, 768);
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
    GL_RENDERBUFFER, nShadowPassRBOName);

// Begin offscreen rendering
... wglMakeCurrent(g_hDC, hRCMain);

// Setup Main context
UINT nRemoteDataFBOName = 0;
glGenFramebuffers(1, &nRemoteDataFBOName);
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, nRemoteDataFBOName);

UINT nRemoteDataRBOName = 0;
glGenRenderbuffers(1, &nRemoteDataRBOName);
glBindRenderbuffer(GL_RENDERBUFFER, nRemoteDataRBOName);
glRenderbufferStorage(GL_RENDERBUFFER, 1, DEPTH_COMPONENT24, 1024, 768);

glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_RENDERBUFFER, nRemoteDataRBOName);

Once off-screen rendering is complete, the resulting image data can be brought back to the primary GPU for use with the final scene.

// Copy data to RBO on main context
wglBlitContextFramebufferAMD(hRCMain, 0, 0, 1024, 768, 0, 0, 1024, 768, GL_DEPTH_BUFFER_BIT, GL_LINEAR);

Now that the data from the associated context (off-screen) is brought to the local GPU, it can be used locally.

Once finished, cleanup all contexts.

// Cleanup contexts
wglMakeCurrent(g_hDC, NULL);
wglDeleteAssociatedContext(hOffScrCtx);
wglDeleteContext(hRCMain);

Synchronizing Data Transfers Between Contexts

OpenGL 3.2 added sync objects and fences to help multiple context applications synchronize work without having to stall the graphics pipeline, among other reasons. This mechanism works well with WGL_AMD_GPU_association, allowing applications to synchronize the rendering and transfer of remote data. It can also be used on earlier versions of OpenGL if the extension GL_ARB_sync is supported.

To use sync objects in the example above, just insert a sync object after remote rendering.

wglMakeAssociatedContextCurrentAMD(hOffScrCtx);

// Render to FBO
// Copy result to main context
wglBlitContextFramebufferAMD(hRCMain, 0, 0, 1024, 768,
0, 0, 1024,768,
GL_DEPTH_BUFFER_BIT, GL_LINEAR);

// Insert Fence
UINT remoteFence = glFenceSync(GL_SYNC_GPU_COMMANDS_COMPLETE, 0);

Then in the main thread, or a thread created for reading the result of the sync object, test the status of
the fence to find out if the data is ready.

// Main rendering loop
...

// Test fence
GLenum syncResult = glClientWaitSync(remoteFence,
GL_SYNC_FLUSH_COMMAND_BIT, 0);

if (syncResult == GL_CONDITION_SATISFIED ||
syncResult == GL_ALREADY_SIGNALED)
{
    // Rendering complete and result ready
}
else if (syncResult == GL_TIMEOUT_EXPIRED ||
syncResult == GL_WAIT_FAILED)
{
    // Error ocured
}

Once finished with sync object, delete it.

    // Cleanup fence
    glDeleteSync(syncResult);

The cost in time to copy data from one GPU to another is not insignificant. Because of this, it is
important to plan what rendering should be done on remote GPUs, leaving time for copies to the main
GPU.

**Techniques to Efficiently Distribute Work**

There are several well known techniques to distribute the rendering on multiple GPUs and to combine
the results to a final image. Examples are

- 2D Decomposition (sort first)
- Database Decomposition (sort last)
- Time based Decomposition
- Eye decomposition for stereo rendering

All of them can be implemented by using the `WGL_AMD_GPU_association` extension. To efficiently implement those techniques on multiple GPUs in a system the application needs to create one rendering thread per GPU. Each GPU can render its portion of the final image and when finished the different sub-images are combined in a compositing step. Typically the rendering threads will look like shown below.

![Diagram](image)

Depending on the technique the data that is blitted and also the composing step will differ.

The following example will show how to implement the DB Decomposition using `WGL_AMD_GPU_association`. 
Example Database Decomposition

The idea of the database decomposition is to distribute the geometry on multiple GPUs and to compose the final image by taking into account the depth values. Each GPU will only render a subset of the total geometry and will provide a color and a depth texture to the composing shader. Depending on the depth value the shader will choose which texel to display.

**The Master Thread:**

First create 2 FBOs each with a color and depth attachment. One is used for local rendering, the second as destination for the blit of the remote GPU.

Render one half of the geometry into fbo[0]:

```c
// Draw to local FBO
glBindFramebufferEXT(GL_FRAMEBUFFER, fbo[0]);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glPushMatrix();
glRotatef( angle, 0.0f, 1.0f, 0.0f);
// Draw only half of the elements
ra->drawRange(0, ra->getNumElements()/2);
glPopMatrix();
// Bind fbo[1] as destination for the blit.
glBindFramebufferEXT(GL_DRAW_FRAMEBUFFER, fbo[1]);
// Update rotation angle for the next frame
angle += 0.05;
// Indicate that master is ready -> Slave will start Blit
ReleaseSemaphore(gMasterReady, 1, 0);
// Wait for slave to finish blit
WaitForSingleObject(gSlaveReady, INFINITE);
// Compose Results
glBindFramebufferEXT(GL_FRAMEBUFFER, 0);
if (gReadyToCompose)
    compose(pWin->getWidth(), pWin->getHeight(), ct, dt);
pWin->SwapBuffer();
gReadyToCompose = false;
```

**The Slave Thread:**

First create a FBO as render target.
Render the second half of the geometry into FBO. As soon as the Master has also finished rendering trigger the blit and release Semaphore when ready.

```c
// Draw to local FBO
glBindFramebufferEXT(GL_FRAMEBUFFER, fbo);

// Draw scene into fbo that is bound to ac
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

glPushMatrix();

// Draw the second half of the elements
ra->drawRange(ra->getNumElements()/2, ra->getNumElements()/2);

glPopMatrix();

// Wait for Master to be ready
WaitForSingleObject(gMasterReady, INFINITE);

// Blit into FBO on master GPU
wglMakeAssociatedContextCurrentAMD(AssociatedGLRC);
wglBlitContextFramebufferAMD(GLRC, 0, 0, w, h, 0, 0, w, h, GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT, GL_NEAREST);

// Insert fence in gl stream to check when the Blit is done
BlitReadyFence = glFenceSync(GL_SYNC_GPU_COMMANDS_COMPLETE, 0);

// Wait for blit to finish
GLenum BlitStatus = glClientWaitSync(BlitReadyFence, GL_SYNC_FLUSH_COMMANDS_BIT, maxTimeout);

if (BlitStatus == GL_CONDITION_SATISFIED || BlitStatus == GL_ALREADY_SIGNALED)
gReadyToCompose = true;
else
    gReadyToCompose = false;

// Indicate that blit is finished
ReleaseSemaphore(gSlaveReady, 1, 0);

glDeleteSync(BlitReadyFence);
```

**Composing:**

Draw a screen aligned quad with the following fragment shader bound.

```c
varying vec2 Texcoord;

// Textures renderd by master
uniform sampler2D color0;
uniform sampler2D depth0;

// Textures rendered by slave
uniform sampler2D color1;
uniform sampler2D depth1;

void main()
{
    float d0 = texture2D(depth0, Texcoord).r;
    float d1 = texture2D(depth1, Texcoord).r;
    vec4 c0 = texture2D(color0, Texcoord);
    vec4 c1 = texture2D(color1, Texcoord);

    if (d0 < d1)
    {
        gl_FragColor = c0;
    }
    else
```
The pictures below show the different color and depth textures that are used to get the final image. The blue geometry was rendered on GPU1 the red on GPU 2.

This approach can easily be modified to compute shadow maps, environment maps or reflections on the second GPU.
Using Dissimilar GPUs

The ideal situation is to have multiple high performance GPUs. But since the environment applications run in cannot always be controlled, this extension makes efforts to provide applications with as much environmental information as possible. An application can get both the local memory size as well as the speed for all available GPUs in a system.

While an application can’t necessarily change the GPU currently in use driving the display the application is running on, it can control how much and which portions of the rendering process are done on which GPU. For cases where the local GPU is very low power, such as an integrated GPU, all rendering can be done on a remote GPU with the result sent back to the GPU the window resides on. The best option for using dissimilar GPUs may be to throttle use of lower power GPUs in a way they can assist in rendering scenes while still competing off-screen rendering in time for the result to still be helpful. Because every system and GPU configuration may be different, experimentally testing with realistic loads will be the best way to determine what amount this will be.

Interactions with Other Features

ATI CrossFire™

ATI CrossFire and GPU association are two separate methods to accomplish accelerated performance on systems with multiple adaptors. ATI CrossFire mode is enabled by a user, not an application. When ATI CrossFire is enabled, the GPUs tied together through ATI Crossfire can only be addressed as a single GPU. Paired GPUs cannot individually be context targets. The GPU information returned when calling wglGetGPUInfoAMD with the ID of the ATI CrossFire pair will be that of the most capable GPU.

GPU Load Balancing

Users may also enable GPU Load Balancing mode. When running in this configuration, the GPU a context resides on is dynamically selected based on usage parameters. However, creating a GPU associated context overrides the GPU Load Balancing target.

MultiView

Normally, Multiview will cause a context tied to a specific window to be allocated on the GPU which is powering the monitor on which the window resides. When a context is created with the GPU association extension, the context is not tied directly to a specific window. The GPU association extension will override a MultiView bias and allocate the context on the requested GPU.
System Configuration Compatibility
For these extensions to be effective, multiple ATI graphics cards must be present in a system are one time. Additionally, there are some Operating System restrictions that limit when this mode can be used.

Windows® XP, Windows Vista and Windows 7
On Windows operating systems, all GPUs intended for use with WGL_AMD_GPU_association must be visible to the OS. This effectively means the windows desktop must be extended to include at least one display head of each GPU intended for use in remote rendering. No applications or windows need be present on the additional GPU displays.

Availability
The WGL_AMD_GPU_association extension is currently shipping on ATI Radeon™ and ATI FirePro™ hardware drivers as of Catalyst 9.3. It is supported on the following graphics cards:

- Professional Graphics
  - ATI FirePro™ V3700, V3750, V5700, V8700 Series and newer
  - ATI FireGL™ V8600, V7600, V5600, V3600, V7700 Series and newer
- Consumer Graphics
  - ATI Radeon™ HD4800, HD4600, HD 4500, HD4300 Series Graphics and newer
  - ATI Radeon™ HD3800, HD3600, HD3400 Series Graphics and newer
  - ATI Radeon™ HD2900, HD2600, HD2400 Series Graphics and newer

Catalyst drivers can be downloaded from http://ati.amd.com/support/driver.html.

Linux support will be released in the near future.

Conclusions
As graphic technologies advance, becoming more affordable and available, systems with more than one GPU are becoming commonplace. However, harnessing all of this power has still been a challenge. But with the use of extensions such as WGL_AMD_GPU_association, applications can take advantage of many different graphics resources in one system. WGL_AMD_GPU_association also allows applications to decide how to divide and distribute rendering tasks based on rendering load and internal application metrics.

Further Reading

- OpenGL  3.2 (http://www.opengl.org/registry/doc/glspec32.core.20090803.pdf)
- WGL_AMD_GPU_association (http://www.opengl.org/registry/specs/AMD/wgl_gpu_association.txt)
Copying Objects Other Than Render Buffers

OpenGL 3.0 added

The cost in time to copy

GPU Association for Linux

GPU association is also supported on Linux if the extension GLX_AMD_GPU_association is supported. This extension is very similar to WGL_AMD_GPU_association and functions in the same way.

Use glXGetGPUIDsAMD to get the number of supported GPUs.

```c
UINT glXGetGPUIDsAMD(UINT maxCount, UINT *ids);
```

Use glXGetGPUInfoAMD to find out more information about a given GPU.

```c
INT glXGetGPUInfoAMD(UINT id, INT property, GLenum dataType, UINT size, void *data);
```

glXGetContextGPUIDAMD will return the GPU ID a context is executing on.

```c
UINT glXGetContextGPUIDAMD(GLXContext ctx);
```

To create a context that is associated with a specific GPU, call glXCreateAssociatedContextAMD.

```c
GLXContext glXCreateAssociatedContextAMD(UINT id, GLXContext share_context);
```

Use glXCreateAssociatedContextAttribsAMD to create a context with specific attributes that is also associated with a specific GPU.

```c
GLXContext glXCreateAssociatedContextAttribsAMD(UINT id, GLXContext share_context, const int *attrib_list);
```

Once finished with a context, call glXDeleteAssociatedContextAMD.

```c
BOOL glXDeleteAssociatedContextAMD(GLXContext ctx);
```
To use an associated context, call `glXMakeAssociatedContextCurrentAMD`.

```c
BOOL glXMakeAssociatedContextCurrentAMD(GLXContext ctx);
```

To get the handle of the current associated context, call `glXGetCurrentAssociatedContextAMD`.

```c
GLXContext glXGetCurrentAssociatedContextAMD();
```

To get the handle of the current associated context, call `glXBlitContextFramebufferAMD`.

```c
VOID glXBlitContextFramebufferAMD(GLXContext dstCtx,
    GLint srcX0, GLint srcY0,
    GLint srcX1, GLint srcY1,
    GLint dstX0, GLint dstY0,
    GLint dstX1, GLint dstY1,
    GLbitfield mask, GLenum filter);
```

The use of these functions follows the same paradigm as those for the WGL versions described in the previous sections. More information on the details of the GLX interfaces can be found in the `GLX_AMD_GPU_association` extension.