GPU Command Buffer

This are mostly just notes on the GPU command buffer

The GPU Command Buffer system is the way in which Chrome talks to the GPU either OpenGL or OpenGL ES (or OpenGL ES emulated through ANGLE). It is designed to have an API that emulates the OpenGL ES 2.0 API enforcing the restrictions of that API and working around incompatibilities in drivers and platforms.

Goals:

The #1 goal of the command buffer system is security. Graphics systems in OSes have gaping security holes. 2 simple examples are you can allocate a texture or a buffer and the memory returned is left as is. That memory is often left over from other applications and could contain passwords, images or other data that should not be visible to the calling app. Similarly there are many API functions that are buggy or have poorly designed APIs that make them easy to call in ways that would crash the browser. The #1 goal of the GPU process is to prevent these problems.

The #2 goal is compatibility across systems. From the POV of clients there should be no differences in behavior across systems. In some cases that means enforcing restrictions that are not there on the actual system. Examples include disabling advanced GLSL features. Others involve working around bugs by re-writing shaders or other techniques.

The #3 goal is speed. Speed is why a command buffer implementation was chosen. The client can write commands very quickly with little or no communication with the service and only once in a while tell the service it has written more commands. For example, another implementation could have used a separate IPC for each OpenGL ES 2.0 function but that would arguably be too slow. The command buffer gets another speed boost because it effectively parallelizes calls into the OS graphics API. A call like glUniform or glDrawArrays might be a very expensive call but because of the command buffer the client just writes a few bytes into the command buffer and is done. The GPU process calls the real OpenGL function on another process which effectively makes the program multi-core.

Implementation:

The basic implementation is a "command buffer". A client (the render process, pepper plugin, etc..) writes commands into some shared memory. It updates a 'put' pointer through IPC telling the GPU process how far it has written into that buffer. The GPU process or service then read commands from that buffer. For each command it validates the command, its arguments, and whether or not the arguments are appropriate for the current state of the OSes graphics API and only then makes the actual call into the OS. This means even a compromised renderer running native code, writing its own commands, can hopefully not get the GPU process to call the graphics system in such a way as to compromise the system.

When writing new service side code, please keep that in mind. Never design a new command that requires the client to be well behaved. Assume the client can go rouge. For example, make sure the service's bookkeeping will never be wrong no matter what the client does.

API Layers:

Life of a GL call in Chrome

In simple terms:
gl2.h->gles2_c_lib.cc->GLES2Implemetation->GLES2CmdHelper...SharedMemory...->GLES2DecoderImpl->ui/gfx/gl/gl_bindings->OpenGL

There is an interface, CommandBuffer, that is responsible for coordinating communication between GLES2CmdHelper and GLES2DecoderImpl. It has methods for creating and deleting shared memory as well as communicating the current state back and forth. Specifically sending the latest 'put' pointer from the client through AsyncFlush() or Flush() and for getting the lastest 'get' pointer through the results of 'Flush'

An implementation of CommandBuffer called CommandBufferService directly talks to GLES2DecoderImpl. If you had a single threaded single process chrome you could pass an instance of CommandBufferService to GLES2CmdHelper and the ideas is things would just work. In the the real multi-process chrome there is another implementation, ComamdBufferProxy which uses IPC to talk from the client to the service through GpuCommandBufferStub to GpuScheduler to CommandBufferService.

Client side code:

Note: Everything in src/gpu/command_buffer/client and src/gpu/command_buffer/common must compile WITHOUT EXTRA LIBRARIES as they are used in the untrusted Pepper plugin

These define the public OpenGL ES 2.0 interface
src/third_party/khronos/GLES2/gl2.h
src/third_party/khronos/GLES2/gl2ext.h

This defines the C interface. Most of this is auto generated
src/gpu/command_buffer/client/gles2_c_lib.cc
src/gpu/command_buffer/client/gles2_c_lib_autogen.h

This is the actual client side implementation that writes commands into the command buffer. Most of this is auto generated.
src/gpu/command_buffer/client/gles2_implementation.cc
src/gpu/command_buffer/client/gles2_implementation_autogen.h

This is a mostly auto generated class to help with formatting commands.
src/gpu/command_buffer/client/gles2_cmd_helper.h
src/gpu/command_buffer/client/gles2_cmd_helper_autogen.h

These define the actual format of the commands
src/gpu/command_buffer/common/cmd_buffer_common.h
src/gpu/command_buffer/common/gles2_cmd_format.h
src/gpu/command_buffer/common/gles2_cmd_format_autogen.h

Service side code:

This is the code that reads the commands, validates and calls OpenGL.
src/gpu/command_buffer/service/gles2_cmd_decoder.cc
src/gpu/command_buffer/service/gles2_cmd_decoder_autogen.cc

3 ways of transferring data

There are 3 ways of transferring data through the command buffer.

#1) In a command itself

Commands can either have a hard coded length (glUniform4f for example takes exactly a location and 4 floats) or they can have a variable length (glUniform4fv takes N sets of 4 floats). The data is inserted after the command in the command buffer and the length of the command itself is updated to contain that data.

Advantages:
* Easy. Fire and forget

Disadvantages:
* Commands have a maximum length of 1meg - 1
* Commands can only be as long as the command buffer itself.

#2) In shared memory

Some commands transfer data in shared memory. TexImage2D for example, the client puts data into a shared memory. The command itself has an shared memory id and an offset into that shared memory as well as either an explicit or implicit size. For TexImage2D the size is implicit.

Advantages:
* Can transfer any size
* Can pre-allocate the shared memory and fill any time (glMapTexSubImage2D for example)

Disadvantages:
* Must check with the server when it has actually used the contents of the shared memory.

#3) In a bucket

Buckets are a kind of abstractions of #2. You define a bucket by size (1 command) you then transfer the data into the bucket through shared memory (n commands), finally you issue the command you really wanted to issue (ShaderSource, CompressedTeximage2D, ...) and reference the bucket.

The problem buckets attempt to solve is. Imagine you are trying to implement TexImage2D and you only have 1meg of shared memory and you are asked to send a 3meg texture. You can't call TexImage2D with your 3 meg of data since you only have 1 meg so instead you call TexImage2D with no data to define your texture and then call TexSubImage2D 3 times to transfer your data. GLES2Implementation actually does this.

Now imagine you are trying to implement ShaderSource. You are passed a 3 meg string and you only have 1meg of shared memory. There is no ShaderSubSource function so you can't use the previous solution. Instead, you create a bucket of 3 meg, transfer the data to that bucket 1meg a time, then issue the ShaderSource command referencing the bucket.

Advantages:
* Do not need a "SubData" command implemented
* Can handle data larger than shared memory

Disadvantages:
* Slower. Data has to be copied out of shared memory and into a bucket.

Adding a command:

Here are some terse notes for adding a new command:

1. Add your function to src/gpu/command_buffer/cmd_buffer_functions.txt

   Note: See examples. For new enums, GLenumTypeOfEnum. For GLint, GLintptr or GLsizei, if negative is disallowed GLxxxNotNegative. For resources GLidTypeOfResource

2. Add your function to _FUNCTION_INFO in src/gpu/command_buffer/build_gles2_cmd_buffer.py

   Copy a function that is similar to yours.

3. run build_gles2_cmd_buffer.py from the command_buffer directory

   Note: we don't currently run this as part of the build as this lets us easily see the changes during code review.

4. Add your function to src/third_party/gl2ext.h if you want it callable as an OpenGL like function.

   If you only need it callable from WebGraphicsContext3DCommandBufferImpl then you don't need to do this step.

5. If your function adds any new GL ENUMs to other functions (most commonly glGetInteger, glTexImage2D, etc.)

   1. Add them in src/gpu/command_buffer/service/function_info.cc

      In particular, add them only if your functionally is available and if they are requested. See code in function_info.cc

   2. Add them in src/gpu/command_buffer/common/gles2_cmd_utils.cc

      - for glGetInteger enums add them to GLES2Util:GLGetNumValuesReturned
      - for texture and buffer formats see ElementsPerGroup, BytesPerElement
      - for texture and renderbuffer formats see GetChannelsForFormat

6. See these CLs as examples
   - http://codereview.chromium.org/8772033/

Texture issues

In order to prevent a user program from reading uninitialized vram all textures must be cleared before being used. In order to increase the speed of programs that upload a lot of textures this clearing happens lazily. If you call glTexImage(..., null) the command buffer will create the texture level and mark it as unclear. Before any read or write to that texture the command buffer will clear it.

If you add functions that update textures you need to call the code that clears an uncleared level by calling TextureManager::ClearTextureLevel for the level in question. You can see examples of this in GLES2DecoderImpl::DoTexImage2D, GLES2DecoderImpl::DoCopyTexSubImage2D, etc..

OpenGL Quirks to be aware of

- There are N * M texture binding points.

The reference pages for OpenGL ES 2.0 were derived from OpenGL 1.0 written in 1993 when maybe there was only 1 texture allowed and talk about 2 binding points, GL_TEXTURE_2D, and GL_TEXTURE_CUBE_MAP. But, glActiveTexture select the active texture slot, each one has these binding points. To give an example

```
GLuint textures[2];
glGenTextures(2, textures);
texture0 = textures[0];
texture1 = textures[1];
glBindTexture(GL_TEXTURE_2D, texture0);
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, texture1);
glActiveTexture(GL_TEXTURE0);

// This command will effect texture0 not texture1
glTexImage2D(GL_TEXTURE_2D, ....
```

As more texture targets are added there are more binding points.

- glGenXXX only reserves a name, it does not create a resource.

  In other worlds

  ```
  GLuint tex;
  glGenTextures(1, &tex);
  printf("%s\n", glIsTexture(tex) ? "true" : "false");
   // prints false
  glBindTexture(GL_TEXTURE_2D, tex);
  printf("%s\n", glIsTexture(tex) ? "true" : "false");
   // prints true
  ```

  This is a arguably a bug in the command buffer right now though there is code to work around it. You'll see we create internal tracking objects on glGenXXX but they are marked as invalid until glBindXXX time (except for Queries)

- glBindXXX creates objects.

  You do NOT have to call glGenXXX before calling glBindXXX. You can make up any ids you like on your own for buffers, framebuffers, renderbuffers and textures and GL will automatically create an object for you when you call glBindXXX.

  Note: This is not true for Queries, Programs and Shaders.

- Resources are ref counted.

  This has many issues. In general, calling glDeleteXXX does 2 things. It released the 'id' so the 'id' if used again, will generate a new resource. It releases the references in the current context's state and the current context's bound objects BUT NO WHERE ELSE.

  So: glDeleteTextures will free the ids for those textures and clear the references from this contexts texture units and from the current framebuffer object. It will NOT clear the references from other framebuffer objects nor other contexts.

  Programs and Shaders are far more quirky.

- Framebuffer objects can not be shared across contexts.

  First all objects are declared as shared by then Appendix C of the OpenGL spec says that FBOs can no actually be shared.

- Texture id 0 is the default texture.

  There's no such thing as turning off textures in OpenLG ES 2.0. Binding texture id = 0 just binds the default texture. All the normal texture commands can effect this default texture.

OpenGL ES 2.0 incompatibilities

Client side arrays

Client side arrays refers to the ability to store vertex data in client side memory and have OpenGL reference it directly. The command buffer itself does not support this. All vertex data must be put in an OpenGL buffer.

To keep compatibility the client side class, GLES2Implementation, emulates client side arrays by tracking the OpenGL attribute state and at draw time, copying any client side arrays into a buffer, updating the vertex attributes to use this buffer, issuing the draw call, then restoring the vertex attribute state. This is a very slow operation and because there is no way to know when the client has changed any of its data those buffers must be updated with every draw call. For this reason, and because more modern versions of OpenGL require it, client side array emulation is compiled out for everything except Native Client.

GL_FIXED

OpenGL ES 2.0 is required to support GL_FIXED as an attribute type (an argument to glVertexAttribPointer). Desktop GPUs do not support this.

The command buffer has optional support for this. Turning it on requires calling glEnableFeatureCHROMIUM("pepper3d_support_fixed_attribs"); as one of the first calls into GL. This makes the command buffer keep it's own copy of all GL buffers. At DrawXXX time, any attributes that are of type GL_FIXED are pulled out of their respective buffers, converted to float and copied to a temp buffer. The attributes are changed to point to this temp buffer. Then the draw happens and the attributes are reset to their previous state.

Clearly this is slow and requires lots of memory. It is there solely to help port OpenGL ES 2.0 apps to NaCl and to pass the OpenGL ES 2.0 conformance tests.

Refactoring Ideas:

Separating decoding the command buffer from emulating OpenGL ES 2.0

Currently those 2 responsibilities are mixed together in GLES2DecoderImpl. There has been some discussion of separating them. A few issues off the top of my head

#1) Validating shared memory

As one example the command for TexImage2D gets passed a shared memory id, an offset and a size. Before the real glTexImage2D is called the service needs to validate that the id is a valid shared memory id, that the offset and size are whole contained inside that shared memory, that the call go glTexImage2D is going to only reference memory instead that shared memory region. To do that requires potentially knowing various state that would normally be not efficiently query-able given a separated OpenGL ES emulation. It's possible the needed state could be easily exposed through separate functions or else maybe changing TexImage2D and similar commands so the size is explicit 'size' instead of implicit 'width * height * type * format'.

#2) Dealing with resource ids

OpenGL ES 2.0 uses int ids for resources. The client uses one set of ids and the service a different set. A mapping from a client id to the service id is kept by the service. In order to avoid a round trip from client to server to manage those ids, for clients context that are not sharing resources, the ids are completely managed on the client and just their usage is communicated to the service. The service makes up a service id to associate with a given client id as needed. Under the current design this works. If the command buffer code was separated from the OpenGL ES 2.0 emulation code a new method of managing these ids would need to be inserted, possibly requiring a double mapping, mapping a client id to a command buffer service id and mapping a command buffer service id to the OpenGL ES 2.0 emulation id. Again, their may be ways to design out that issue.

Moving functionally from GLES2DecoderImpl to the various resource managers

GLES2DecoderImpl is HUGE. Nearly 8000 lines. There's been talk of moving large chunks of functionality to the various resource managers. For example, move all handling of the texture functions, TexImage2D, TexSubImage2D, TexParameter, CopyTexImage2D, CopyTexSubImage2D, CompressedTexImage2D, CompressedTexSubImage2D, GenTexture, DeleteTexture, IsTexture, TexStorage2DEXT from GLES2DecoderImpl to TextureManager.

I'd love to see that happen. Unfortunately I expect it's not a small amount of work. In particular fixing up all the unit tests

Still, it seems like it would be a much cleaner implementation to go that route.

Separate command generation from OpenGL ES

build_gles2_cmd_buffer.py has a nearly 1 to 1 mapping of OpenGL ES functions to commands. Ideally the commands in the command buffer would be separate from the OpenGL api so that it would be easier to add any command needed and not have to expose it as an OpenGL ES extension.

Remove legacy code

Originally the command buffer commands were going to be the public API to the gpu process with the OpenGL ES API as a wrapper. Most game consoles allow you to work directly with command buffers which is one reason for their performance. Being able to work directly with command buffers means you can pre-compute command buffers and patch them on the fly as needed which in turn means your code and do the minimal amount of work and therefore gain a lot of speed. Eventually it was decided not to expose the command buffer commands directly but there is still code based on the original design that could be removed.

low-level commands

Implemented in src/gpu/command_buffer/client/cmd_buffer_helper.cc, src/gpu/command_buffer/common/cmd_buffer_common.h and src/gpu/command_buffer/service/common_decoder.cc are functions that implement

JUMP, CALL and RETURN. These functions intern influence some of the design constraints of the rest of the system. They are not needed unless command buffers are a public interface and could be removed.

3 types of commands

build_gles2_cmd_buffer.py generates 3 versions of many functions. One for each of the data transfer modes above. For example TexImage2D, TexImage2DImmediate and TexImage2DBucket are respectively the transfer buffer implementation of TexImage2D, the data-in-the-command-buffer version of TexImage2D and the bucket version of TexImage2D. When the command buffer was the public interface it seemed important to have all 3 as they each have their pluses and minuses. Now though only the commands used by GLES2Implementation are needed. Maybe the code that generates all 3 versions should be retired.

remove _CMD_ID_TABLE

in build_gles2_cmd_buffer.py _CMD_ID_TABLE's sole purpose is to make sure the ids of commands do not change. This was important when commands were going to be a public API. It no longer matters and can be removed and commands and change ids any time.

size in entries

Left over from the O3D code, the command buffer works on CommandBufferEntry units. Each unit is 32 bits and sizes of commands and command data is calculated in those units. There's a lot of superfluous math involved in converting to an from those units. If instead the code was refactored so that the size of commands was in bytes all of that extra math code could disappear.

Comments

You do not have permission to add comments.