NVISION 08
THE WORLD OF VISUAL COMPUTING
OpenGL and the Future
Michael Gold, Mark Kilgard and Barthold Lichtenbelt
© 2008 NVIDIA Corporation.
NVIDIA

Agenda

- OpenGL 3.0 / GLSL 1.30 (Barthold Lichtenbelt)
  - blichtenbelt@nvidia.com
- OpenGL 3.0 and Cg 2.1 (Mark Kilgard)
  - mjk@nvidia.com
- CUDA <-> OpenGL interop (Michael Gold)
  - gold@nvidia.com

© 2008 NVIDIA Corporation. NVIDIA

Introduce Mark, Michael

Poll: Who is a software developer or works for a software company? Who’s in management?

Who knows what the OpenGL ARB standards body is?

Mention driver developers in the room. Because of time this will be fairly high level, feel free to come talk to us afterwards

OpenGL 3.0

- Announced two weeks ago
- Support for latest generations of Programmable Hardware
  - Installed base > 60 Million units
- New deprecation model with profiles
  - Streamline the API
- Full interoperability with OpenCL and CUDA
  - Access to compute
- Collaboration among hardware vendors and software vendors
  - Solving real needs
- Cross platform
  - Windows XP and Vista, Linux, Mac OS, …

The OpenGL ARB announced a new version of OpenGL and the OpenGL Shading Language two weeks ago. Specifications for both are available right now for download. I will briefly summarize the major new features.

GL 3.0 exposes functionality available in shipping hardware since late 2006. This includes functionality that was as of today only available in extensions from various vendors. OpenGL 3.0 enables over 60 million GPUs already in people's hands. Furthermore, there is a new

Deprecation model: Allows OpenGL to mature without breaking backwards compatibility. The deprecation model offers a path for streamlining the API by removing functionality through a defined multi-step process. I will provide more details later on this and profiles later.

Interoperability with OpenCL and CUDA: OpenCL is the upcoming new Khronos API for compute using a GPU and a CPU. CUDA, as you probably know, is NVIDIA's offering for compute, available today and is support on all OpenGL 3 class hardware from NVIDIA. The ARB decided explicitly to not extend OpenGL to evolve into a general compute engine. OpenGL is for graphics, CUDA and OpenCL for compute. These APIs will be able to share data (like buffer objects) to transfer information back and forth. Michael will talk about this later.

OpenGL 3.0 is defined through collaboration among a large list of hardware and software vendors, including AMD, ARM, Apple, Intel, NVIDIA, S3 Graphics, Blizzard and Transgaming. We're very happy with the input, and continued input, from software vendors Blizzard and Transgaming.

As always, OpenGL was and is cross platform. If you are developing an application that needs to run on a variety of operating systems, OpenGL is your API.

OpenGL 3.0 new features

- Forward-looking context
- Greater VBO performance
- FBO and related extensions
- Conditional rendering
- Transform feedback
- FP internal formats for textures, renderbuffers
- Half-float (16-bit) vertex and pixel data formats
- Array textures
- One and two-channel (R and RG) internal formats for textures and renderbuffers
- RGTC internal compressed texture formats, packed float and texture shared exponent
- sRGB framebuffer support

There are over two dozen new features in OpenGL 3.0 some of the major new features I've listed here. Mark will go over these in more detail in his presentation later. But I wanted to point out three of them specifically.

As I just mentioned, we now have a deprecation model, and part of that is marking functionality deprecated what will be removed from a future version.

Deprecated does not mean removed from the API, it means marked for future removal. If you create a forward-looking context in OpenGL 3.0 today you'll get a context with all the deprecated functionality removed. It provides a preview of the functionality that'll be in a future OpenGL version. More on that in a bit.

A new Vertex Array Object that encapsulate vertex array state like format, type, stride, storage information for each array. It is now also possible to map sub ranges of buffer objects into client space and have more control over flushing parts of that buffer object data for greater performance. You can now also opt out of the normal blocking behavior of the mapbuffer API call.

Not only is the framebuffer_object extension now part of the core, the FBO improvements contain all the functionality in the framebuffer_blit, framebuffer_multisample, packed_Depth_stencil and framebuffer_sRGB extensions. Furthermore, attachments of different width and height or different formats are allowed.

Conditional rendering lets you discard rendering based on an occlusion query

Transform feedback lets you render data into a buffer object, then re-use that buffer object as the source for another rendering pass

FP support for textures and renderbuffers enables you to render to and from FP format data.

Half float vertex array data formats: Input vertex data in a compact float format

Plus, the ability to render to and from single or double channel data.

RGTC texture compression is a new format for 2D images without borders.

And sRGB framebuffer support lets you store data in sRGB format in the framebuffer, including blending support.

GLSL 1.30

- Native integer support
  - bitwise operators, texture return values, uniforms, shader IO
- Expanded texturing support
  - Size queries, offsets, explicit LOD and derivative control, texture arrays, integer support
- Switch statements
- Several new built-in functions
  - Hyperbolic trig functions
  - trunc(), round(), roundEven(), isnan(), isinf(), modf()
  - Integer related: sign(), min/max(), abs(), ...
- Pre-processor token pasting (##)
- User-defined fragment outputs
- Non-perspective interpolation of varyings
- gl_VertexID vertex shader input

© 2008 NVIDIA Corporation.

A major part of all the new functionality can be found in the new shading language, version 1.30.

The highlights are on the slides. One important thing to point out is that GLSL will follow the same deprecation model as the API.

Highlights include:

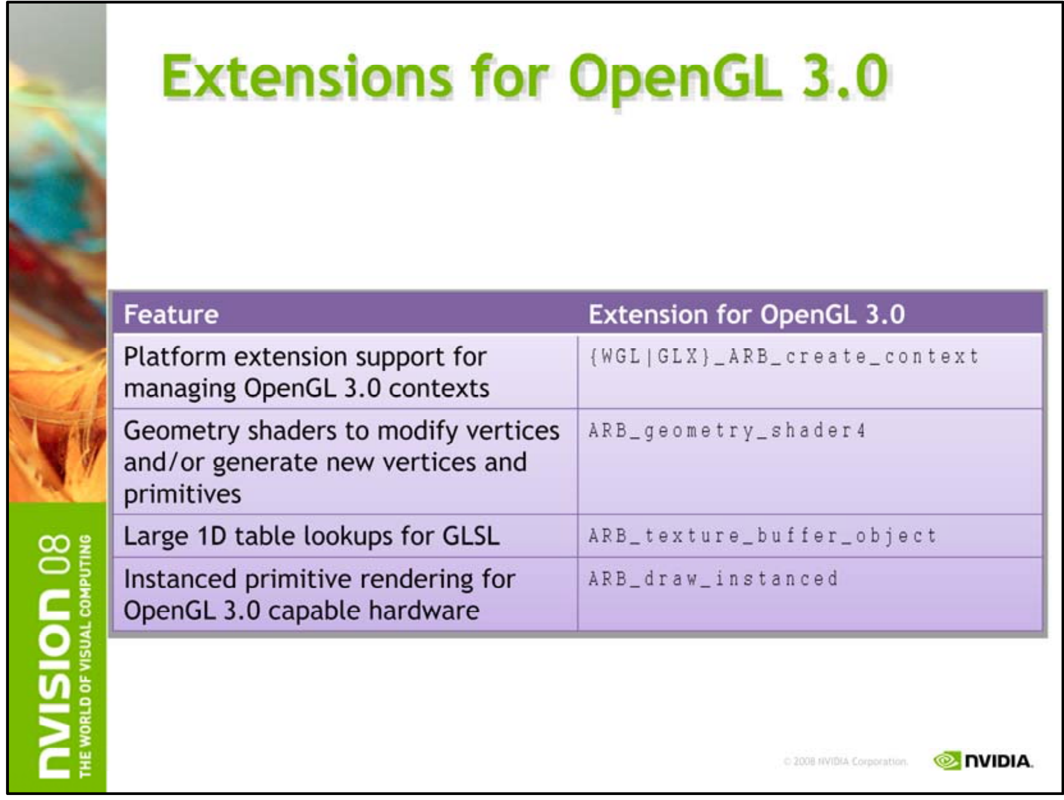
Integers are now at least 32 bits wide, and you can do the usual integer operations on them. Furthermore, I/O in and out of shaders has been enhanced to support integers.

Many more texture functions are added, giving you even more control. Texture arrays are now also part of GLSL 1.30 .

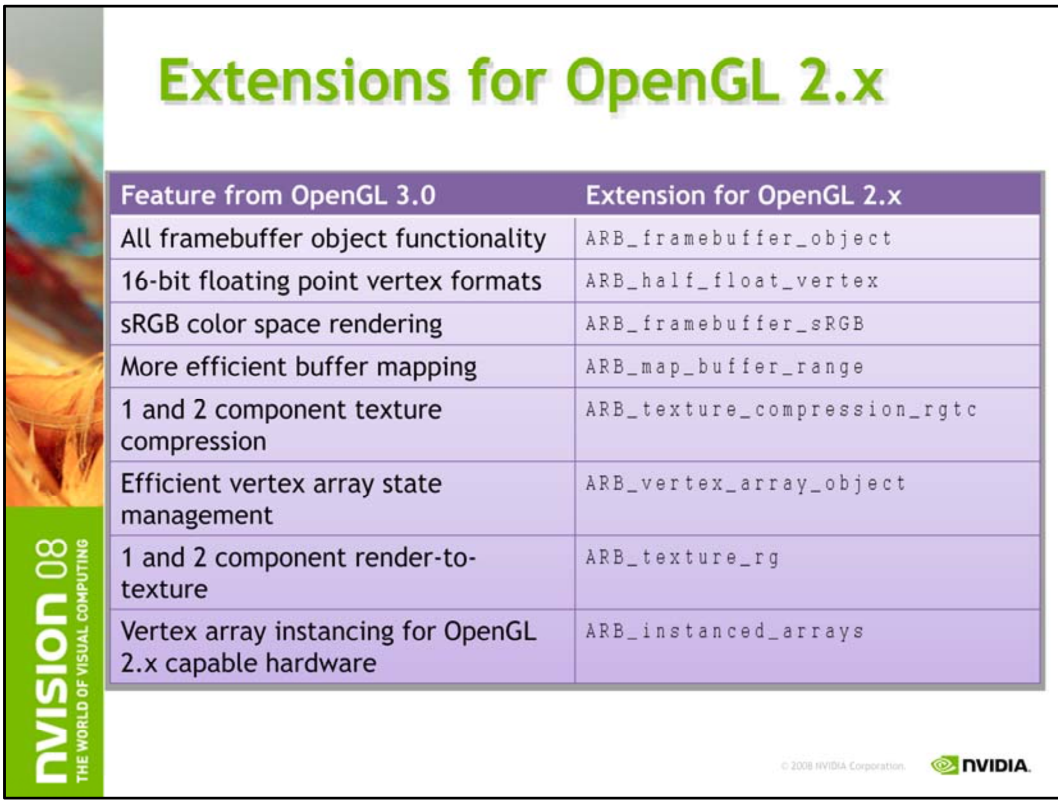
Switch statements were missing and are now added.

On top of that a host of smaller features, listed here on the slide.

There is more than the core OpenGL 3.0 and GLSL 1.30 specifications….



Simultaneous with the OpenGL 3.0 release the ARB has also releasing two sets of extensions. The OpenGL 3.0 extension pack, listed here, and another in the following slide. These are extensions that the ARB felt were almost, but not quite, ready for core inclusion and can be tweaked some more before making it in a future version of the core.



The 2.x extension pack takes functionality that is in OpenGL 3.0 (except for ARB_instanced_arrays) and makes that available for that hardware that cannot quite support all of OpenGL 3.0. As you can see that is quite a lot of stuff.

That includes the vertex array enhancements through VAOs and mapbufferRange, and all the new framebuffer object functionality.

Copyright 2005, All rights reserved

Deprecated features

- **OpenGL has never removed features**
  - Commitment to backwards compatibility is one of OpenGL's strengths
  - After 15+ years, defining new features to work with old features becomes increasingly difficult
- **OpenGL 3.0 does not remove any features**
- **OpenGL 3.0 does mark certain features as deprecated**
  - Redundant, Legacy and obsolete features
  - Parts of OpenGL unlikely to be accelerated
- **Future OpenGL revisions will remove these deprecated features**
  - Guidance to developers to prepare for future revisions
  - Plan to remove these features sooner, rather than later.

One of the new things that the ARB is introducing with OpenGL 3.0 is a path for removal of certain types of functionality. Generally that is functionality that is redundant, slow in practice or subsumed by more modern mechanisms. It is important to note that deprecated means "marked for removal" and not actual removal from the specification. A future revision of the OpenGL specification can remove features marked deprecated. Nothing is removed in OpenGL 3.0.

15+ years of API design makes it increasingly difficult to define how new fucntionality has to work with existing, old, functionality.

Deprecated features

- Fixed-function vertex and fragment processing
- Color-index mode
- Display lists, and Selection and Feedback modes
- GLSL 1.10 and 1.20
- Begin/End based rendering
- Application-generated object names
- Quads and polygon primitives
- Polygon and Line Stipple
- Pixel transfer modes
- Bitmaps, DrawPixels, PixelZoom
- *and quite a few others...*
  - See **Appendix E** of OpenGL 3.0 specification for the list

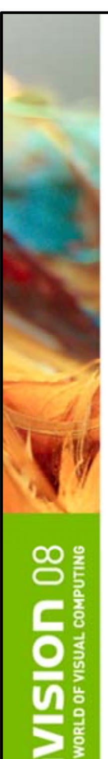
Here is a partial list of features that are marked deprecated in OpenGL 3.0. The full list is in the OpenGL 3.0 specification.

I would like your feedback on this list. What on the list do you absolutely need for your products? Feel free to come talk to us afterwards about this.

Deprecation mechanism

- **Step 1 Core feature**
  - In core, fully supported. **Will** be in the next API version
- **Step 2 Core (Deprecated feature)**
  - In core, marked as deprecated
  - **May** be fully or partly removed in a later version
  - New features need not define interactions with deprecated ones
- **Step 3 ARB approved Extension**
  - **Removed** from core -> an ARB extension (no suffix)
  - Extension spec identifies the removed functionality
  - Vendors may support the extension if markets require it
- **Step 4 Removed from ARB extension list**
  - Could be an EXT or vendor extension, if vendor markets still require it (still no suffixes required)

Deprecation is a four step process. First, a feature is in the core. Next, it is still in the core but clearly marked deprecated in the specification. Next, the ARB will remove it from the core specification and move it into an ARB extension. As you know, extensions are optional to implement by hardware vendors, the core specification is not optional. As a last step the ARB can decide to demote the feature even further, and then it becomes up to a hardware vendor to decide to support the feature as an extension, or not.



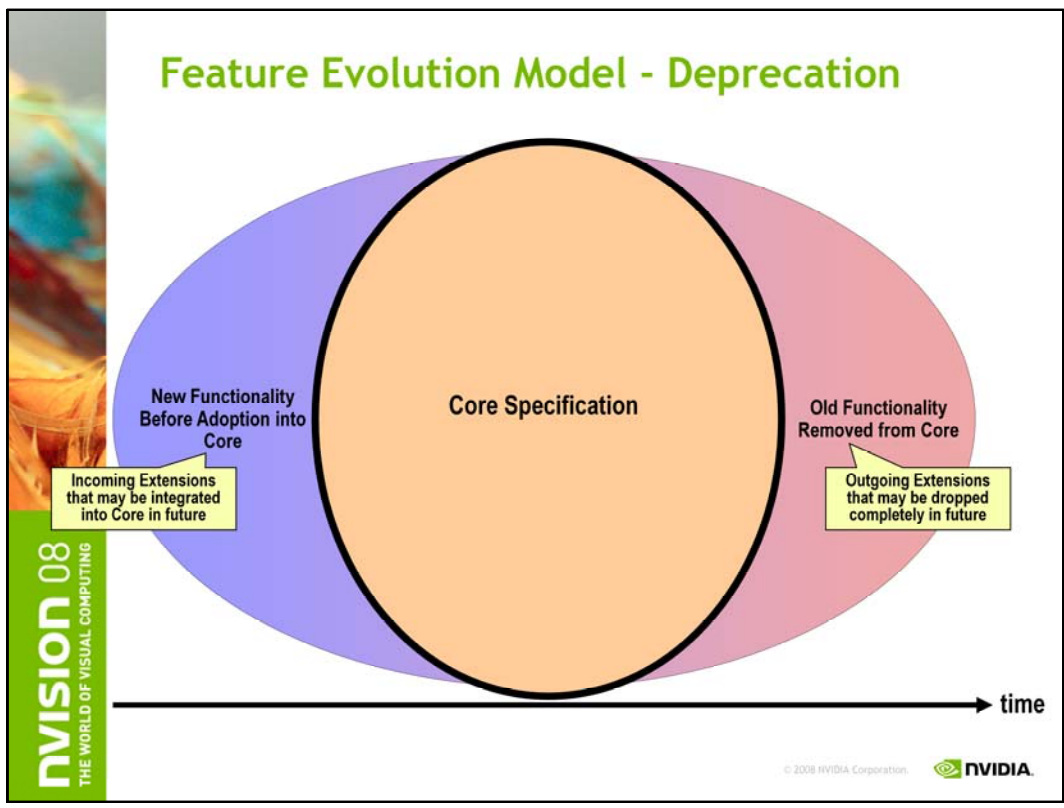
Deprecation mechanism

- Features will be deprecated for at least one spec release (step 2) before being removed
- Extension Path: `Vendor/EXT->ARB->Core`
  - With possible API / functionality changes as we learn from experience
- Deprecation Path: `Core->ARB->EXT/Vendor`
  - No API or functionality changes

© 2008 NVIDIA Corporation.

NVIDIA

Or to say it differently, a feature has a well defined path from extension into the core and out of the core into an extension again. Note that when a feature is on the path into the core, it can change whenever it changes status from ext to arb and then to core. But when it moves out, that is not possible.



Graphically, the lifespan of a feature in OpenGL looks like this. Feature foo first will be an ARB extension, then makes it into the core, possibly with minor modifications to improve upon what is there. Once in the core, the functionality of foo cannot change anymore. Eventually, it can move out of the core and become an extension again.

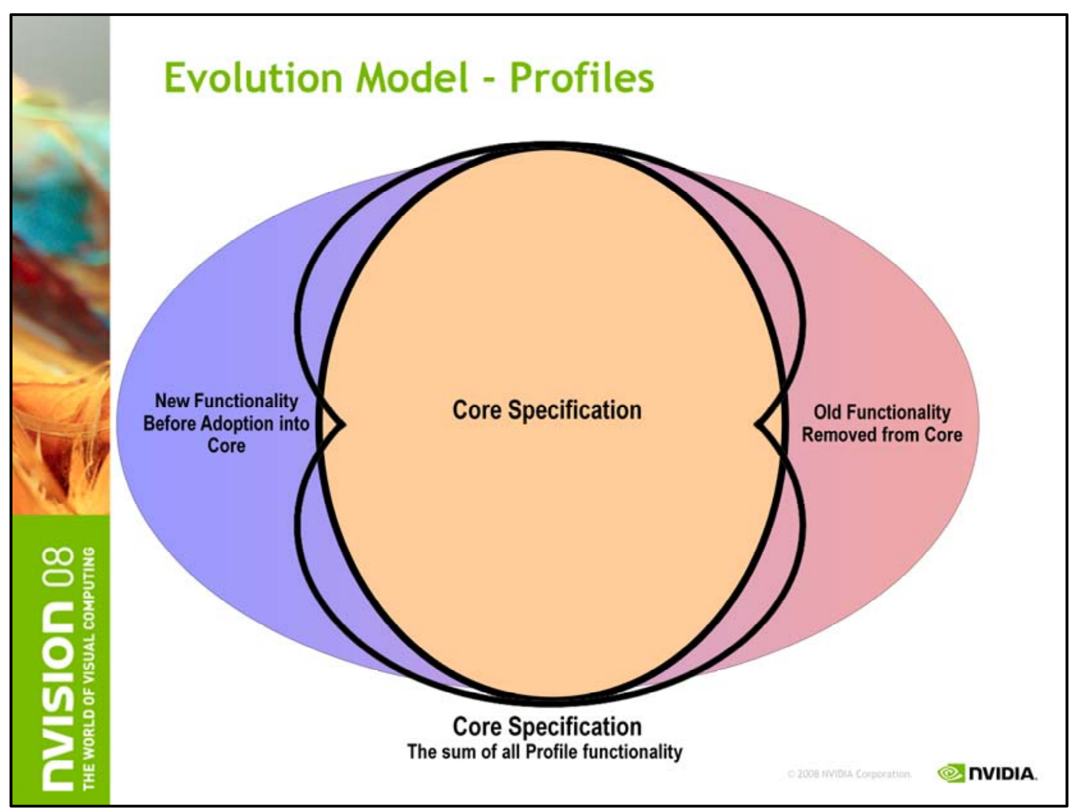
Profiles

- **Encapsulates a set of functionality**
- **Optional to implement for vendors**
- **Sum of all profiles makes up the Core spec**
  - OpenGL 3.0 is one big profile
- **Deprecation mechanism is applied per profile**
- **Only the OpenGL ARB can define profiles**
- **Currently discussing need for "workstation" profile**
  - Could contain most of the deprecated functionality
  - Need input from you!

© 2008 NVIDIA Corporation.

A profile encapsulates functionality needed to meet the needs of a particular market. Conformant OpenGL products may implement one or more Profiles. A Profile is by definition a subset of the whole Core specification. The Core OpenGL specification will contain all functionality in a coherently designed whole. Profiles simply enable products for certain markets to not ship functionality that is not relevant to those markets in a well defined way.

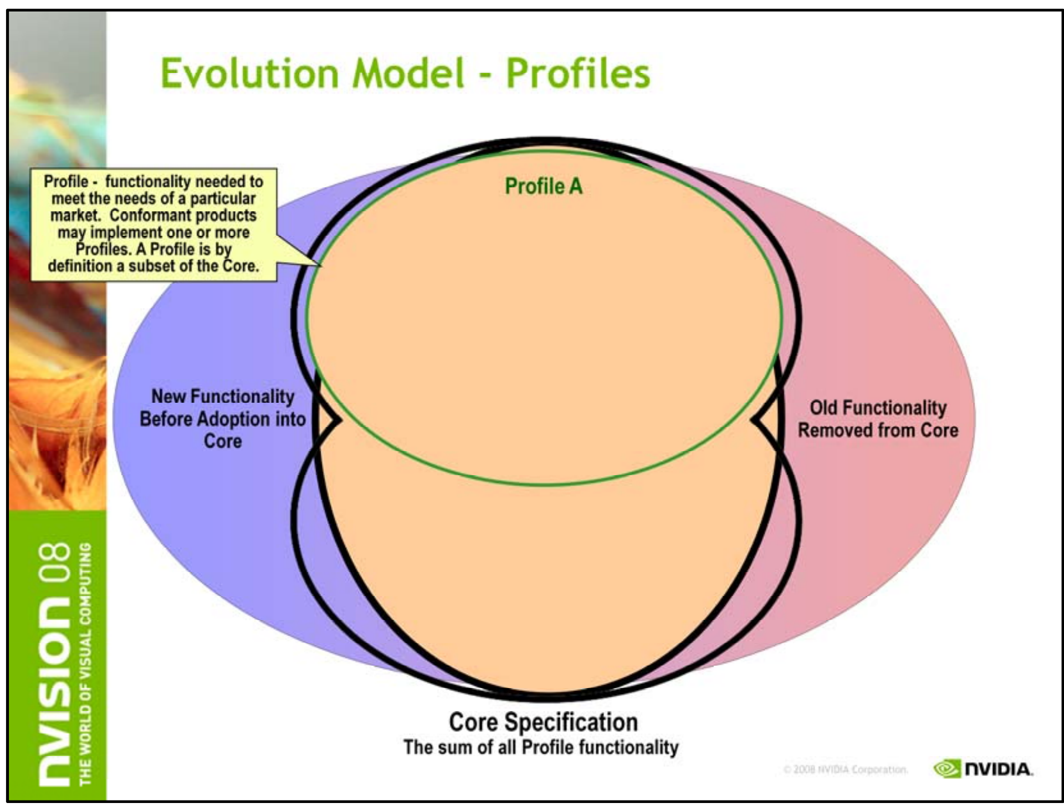
Note that deprecation is applied per profile. Only the ARB can define a profile, in contrast to an extension that any HW vendor can define.



This is a graphical view of profiles. Here you first see one profile, as is the case today With OpenGL 3.0. The thick black line denotes the core specification.

<animate>

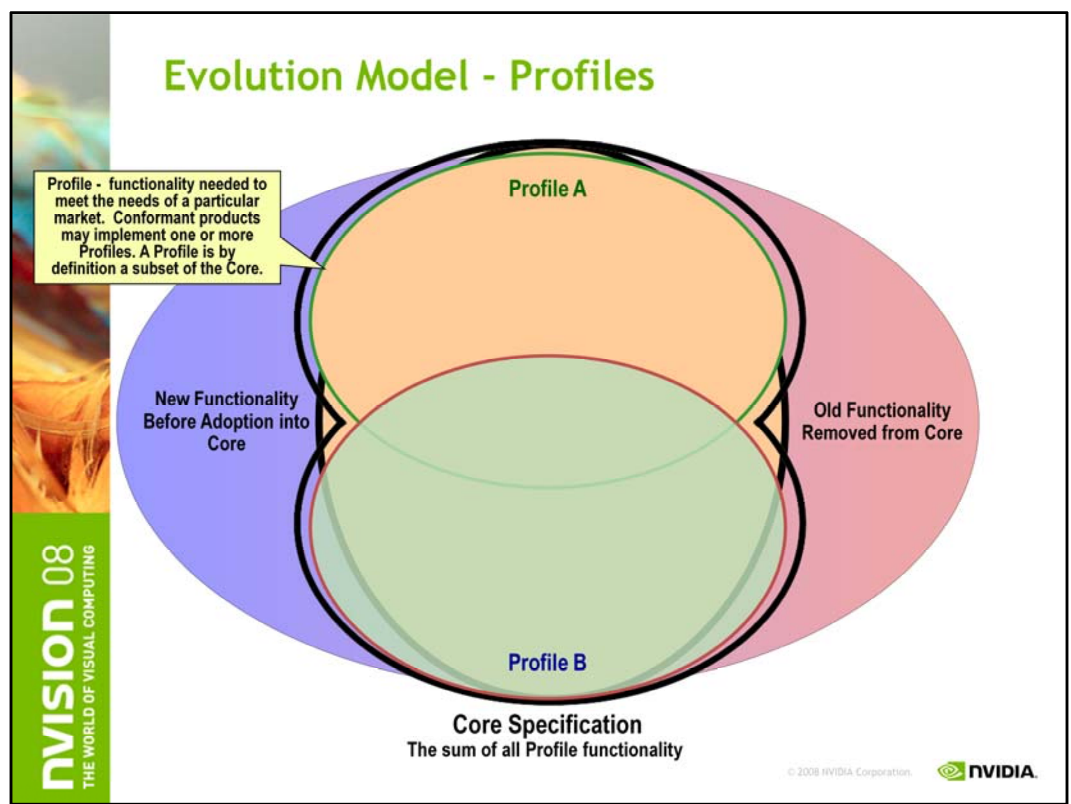
Then an example if there were two profiles. The sum of the two profiles makes up the whole core specification, outlined with the black line. This guarantees that the profiles together form a coherent whole. Profiles cannot define anything that is contradicting another profile.



This is a graphical view of profiles. Here you first see one profile, as is the case today With OpenGL 3.0. The thick black line denotes the core specification.

<animate>

Then an example if there were two profiles. The sum of the two profiles makes up the whole core specification, outlined with the black line. This guarantees that the profiles together form a coherent whole. Profiles cannot define anything that is contradicting another profile.



This is a graphical view of profiles. Here you first see one profile, as is the case today With OpenGL 3.0. The thick black line denotes the core specification.

<animate>

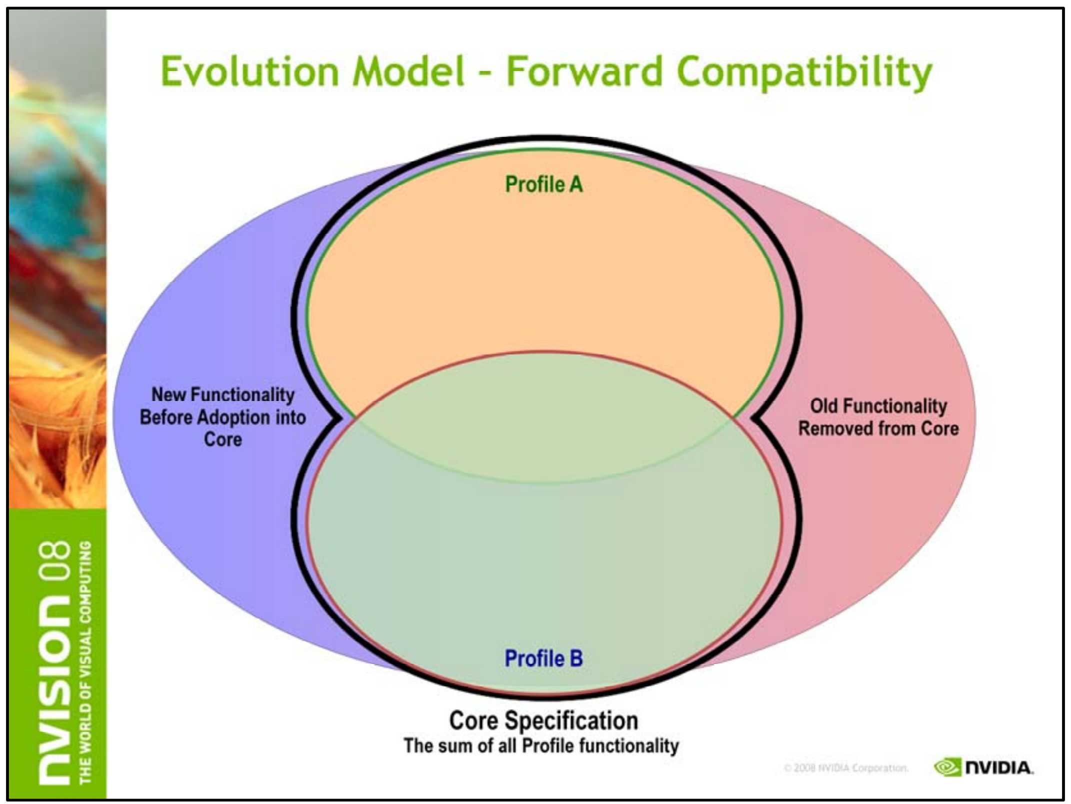
Then an example if there were two profiles. The sum of the two profiles makes up the whole core specification, outlined with the black line. This guarantees that the profiles together form a coherent whole. Profiles cannot define anything that is contradicting another profile.

Context types

- **Full context**
  - Contains all features in a version of the core specification
- **Forward compatible context**
  - Contains only the non-deprecated functionality in a context and profile

© 2008 NVIDIA Corporation.

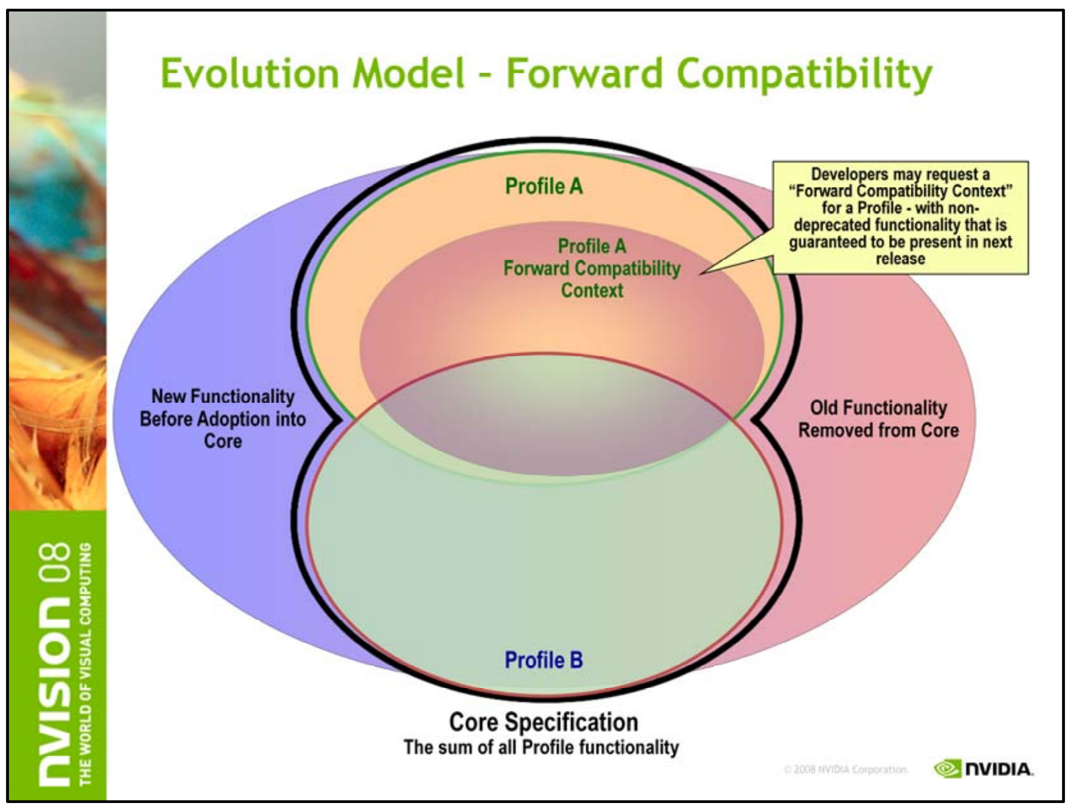
There are two types of contexts that you can create with OpenGL 3.0 and later. A full context and a forward compatible context. The forward compatible one is not a preview. It only shows what is left after the deprecated functionality is taken away. It allows you to develop code today that will work with a future version of OpenGL.



This slide shows the complete picture, with profiles and forward compatible contexts and deprecation. This is an example with two profiles making up the core specification. Each profile can define a forward compatible context, containing all the non-deprecated functionality.

<animate>

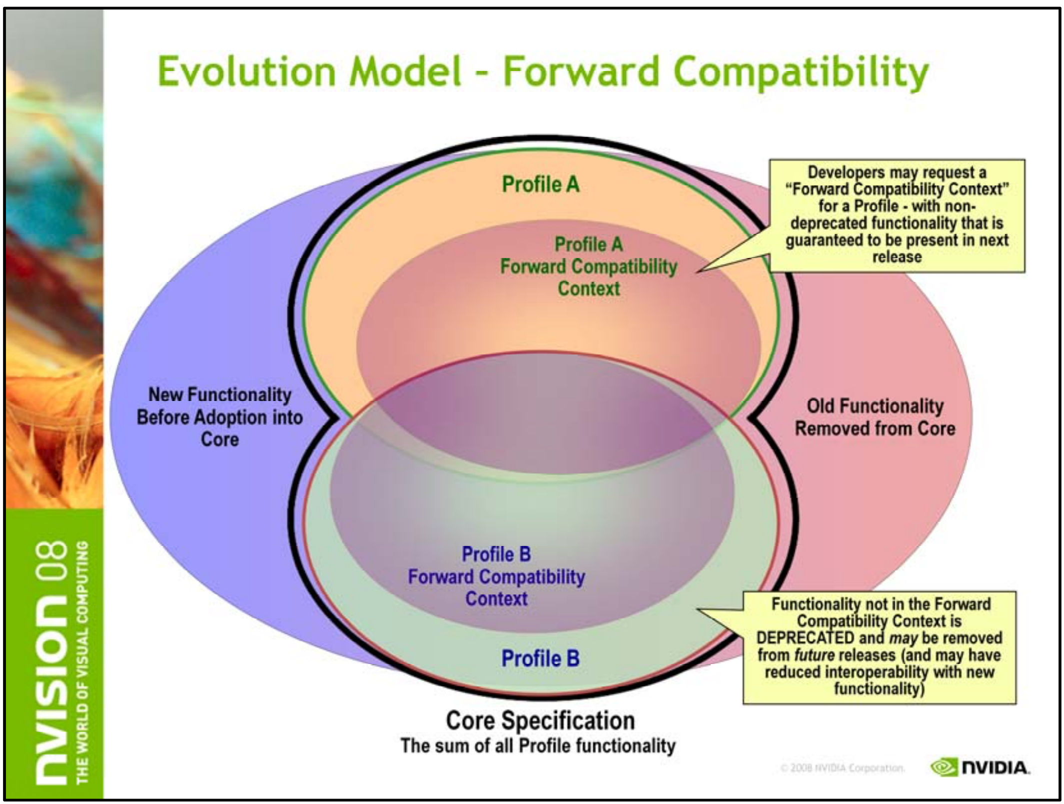
Deprecation, as you can see, is defined per profile.



This slide shows the complete picture, with profiles and forward compatible contexts and deprecation. This is an example with two profiles making up the core specification. Each profile can define a forward compatible context, containing all the non-deprecated functionality.

<animate>

Deprecation, as you can see, is defined per profile.



This slide shows the complete picture, with profiles and forward compatible contexts and deprecation. This is an example with two profiles making up the core specification. Each profile can define a forward compatible context, containing all the non-deprecated functionality.

<animate>

Deprecation, as you can see, is defined per profile.

Context creation

- In the past creating a context gave you whatever version the driver decided
  - No issue since the API was always backwards compatible,
- Starting with OpenGL 3.1, backwards compatibility may no longer exist
  - due to deprecation
  - Apps need a way to specify which functionality they require when creating a context
- Existing context creation calls cannot return 3.0 or later contexts
- WGL/GLX_ARB_create_context
  - To request specific context version, profile, forward compatible context or debug context.
  - `wgl/glxCreateContextAttribsARB()`

© 2008 NVIDIA Corporation.

So, how do I create a full or forward-compatible context?

The new WGL/GLX_ARB_create_context extension introduces a new entrypoint to do exactly this. It takes a list of attributes to describe exactly what version and type of context you want.

This entrypoint has to be used to create an OpenGL 3.0 or higher context. It can be used to create an older context, but the currently existing WGL/GLX methods will still work.

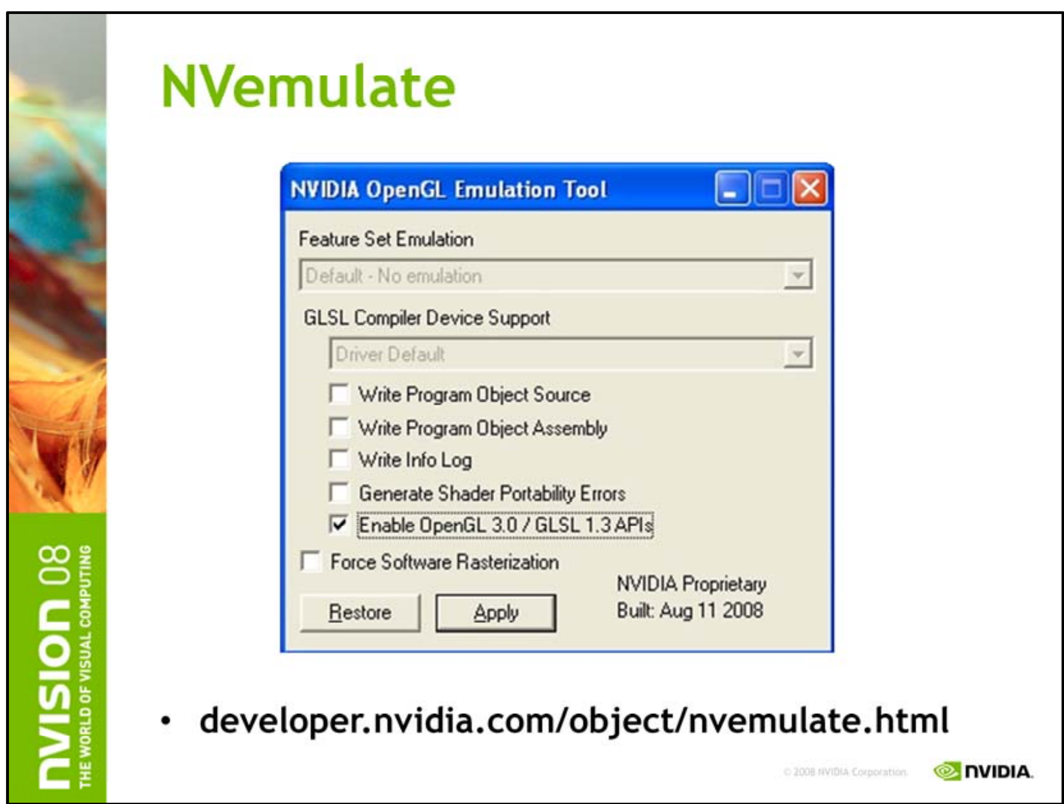
Thus as a developer you have to "opt in" to use the new functionality. One strategy you could follow is to leave any existing code alone, and create another OpenGL 3.0 context for any new code you might want to write.

OpenGL 3.0 beta drivers

- **Beta drivers available for download now**
  - For Windows XP and Vista
  - Linux to follow shortly
  - G80 and higher GPUs supported. Geforce and Quadro
- **Beta drivers, aimed at developers to get started**
- **Supports full OpenGL 3.0 context**
- **Supports GLSL 1.30**
- **Also supporting most of the extensions**
- **See driver release notes for details**

developer.nvidia.com/object/opengl_3_driver.html

NVIDIA has released beta drivers a few weeks ago that you can use to experiment with OpenGL 3.0. Currently we support Windows, but Linux drivers will follow shortly. These are aimed at developers that want to get started. They are beta because we didn't implement everything quite yet and aren't quite as well tested as normally released drivers. Please don't ship an application based on these!



Because the drivers are in beta form, you need Nvemulate to explicitly turn on OpenGL 3.0 and GLSL 1.30 support. IF you don't you'll get OpenGL 2.1 and GLSL 1.20.

Future OpenGL plans

- Schedule driven
- ARB extensions are candidates for folding into a future core
  - ARB_draw_instanced
  - ARB_geometry_shader
  - ARB_texture_buffer_object
- Backing uniform variables with buffer objects
- #include mechanism for GLSL
- Attribute index offsets
- Remove deprecated features
- Profiles
- Object model improvements
- Other functionality you need?

© 2008 NVIDIA Corporation.

In January of this year the ARB set the goal to release OpenGL 3.0 and GLSL 1.30 in its current form, available for download at Siggraph. Because of the Khronos process we need to follow, that meant the specifications needed to be finalized a month earlier. That month gives the Khronos Board of Promoters time to examine the specification, make sure the ARB did do due diligence and gives Khronos members time to sort out any IP issues. At the end of the 30 day period, the spec then gets approved or rejected by the Khronos promoters. In our case, it was approved.

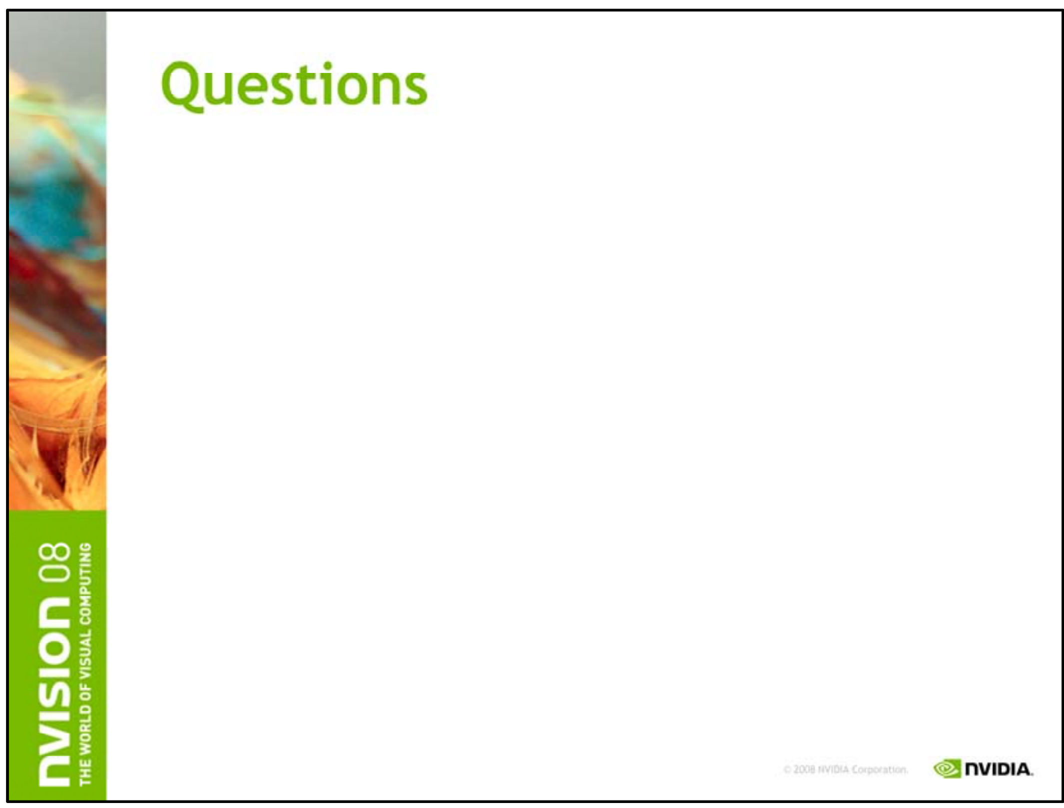
We're going to do this again. We'll be schedule driven in releasing a specification. If a feature doesn't get done in time to meet the schedule, it'll go in the next release. That is what happened with geometry shaders, for example. We like geometry shaders, we want them to be part of the core, but we didn't quite get all the little details sorted out in time for this release. Thus it is on the list for the next release. Similar for the other features on this slide, they are all candidates for a next release, if we can fit it in the schedule.

At the end of September is our next f2f meeting where we'll finalize the schedule for OpenGL 3.1. I expect this to be a fairly short cycle, hopefully well before Siggraph next year.

We'll be taking a hard look at the OpenGL 3.0 ARB extensions and will fold those into the next revision if at all possible.

One thing you might have noticed is missing is a way to back uniform storage with a buffer object. That is high on our priority list. Another item is #include support in the shading language, very handy feature. Then finally

We might define one or two profiles to better support key markets for OpenGL. Initial discussions suggest there might be a need for an entertainment profile and a workstation profile. Profiles are defined by the ARB. Individual vendors are not allowed to define one.



Want your input on deprecated features, profiles, your needs.