Future Directions

Overview Discussion
September 2001

Agenda

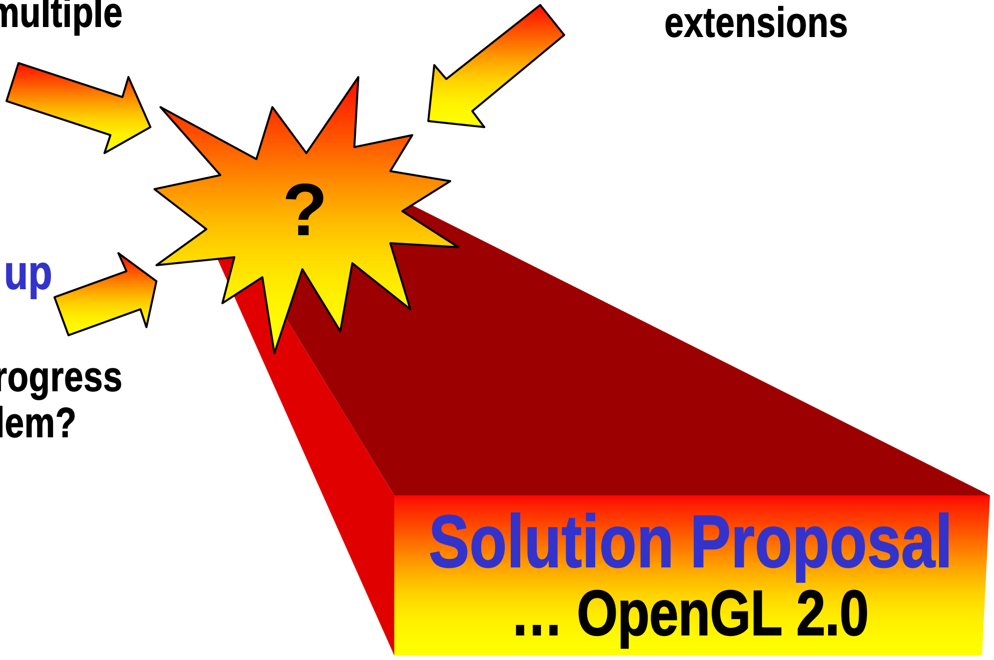
Cooperative discussion for the future of OpenGL

- **Setting the scene**
  - Why we think OpenGL needs to quickly make significant forward progress
- **The outline of 3Dlabs' proposal for a direction for OpenGL**
  - A technical heads-up and a suggested cooperative framework
- **A technical discussion of the "OpenGL 2.0" proposal**
  - Simplifying OpenGL with backwards compatibility
  - Programmability
  - Time Control
  - Memory Management
- **Convergence between Khronos and the ARB**
  - OpenML 1.1 and OpenGL 2.0 are solving many similar problems
- **Open discussion**
  - We need feedback from the ARB
  - Is this the right direction?
  - How can we cooperate to get it done?

OpenGL in Crisis?

Many think OpenGL is not moving fast enough

- **The industry needs OpenGL to be vigorous and evolving**
  - A cross platform API – Windows, Mac, Linux, Embedded
  - Open, multi-vendor standard that cares about backwards compatibility and stability
- **But OpenGL faces a number of critical issues:**



OpenGL is not providing hardware independent access to programmable processors

Current direction is to expose multiple hardware architectures

OpenGL is becoming more and more complex and unwieldy

A confusing mixture of specialized extensions

IP discussions are holding up progress at the ARB

Are IP issues a CAUSE of lack of progress or a SYMPTOM of a deeper problem?

?

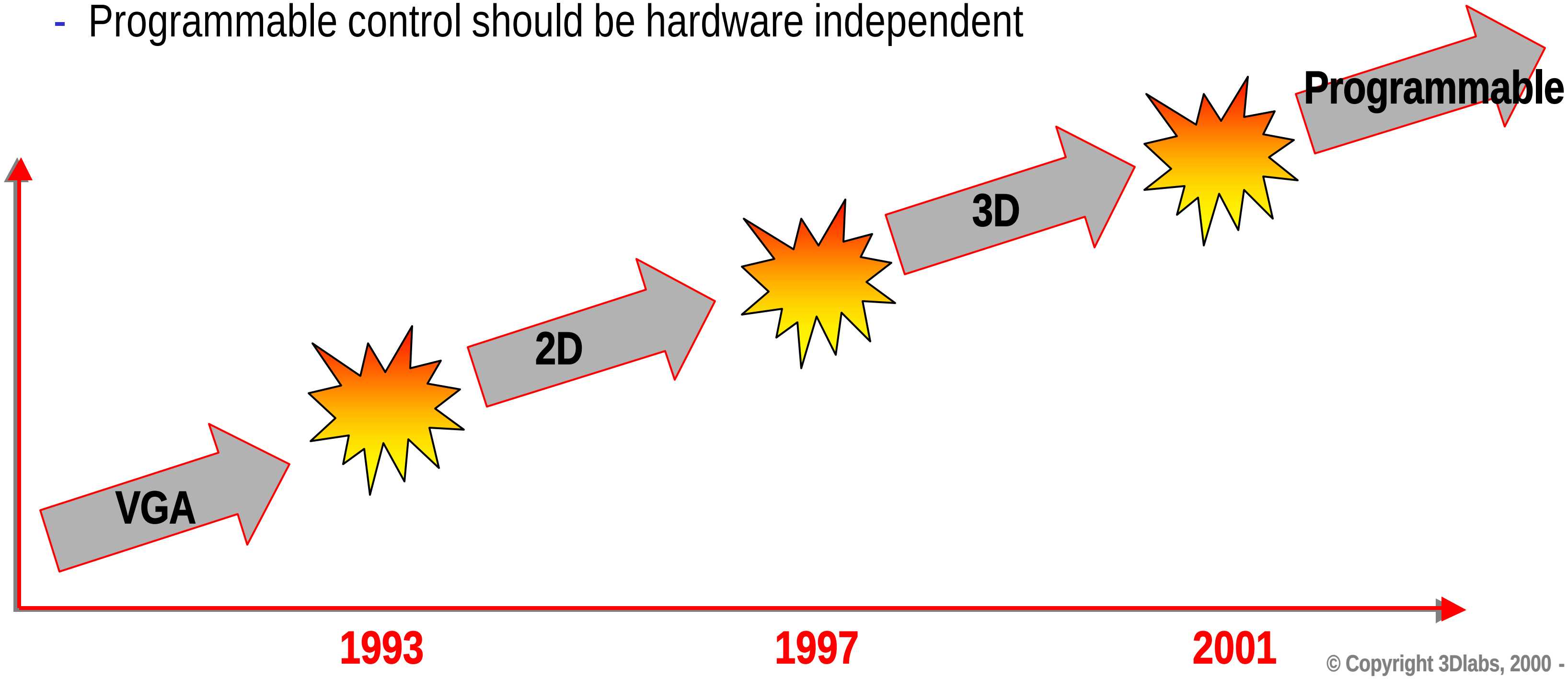
Solution Proposal

... OpenGL 2.0

The Transition to Programmability

A significant industry discontinuity

- **Programmability is becoming recognized as the future of graphics**
  - A lot of work (e.g. at Stanford) on high-level shading languages
- **An Opportunity**
  - Enables new classes of applications
  - Replaces escalating arbitrary complexity with generalized programmability
- **A Challenge**
  - Growing perception that OpenGL is lagging Direct3D in programmable functionality
  - Programmable control should be hardware independent



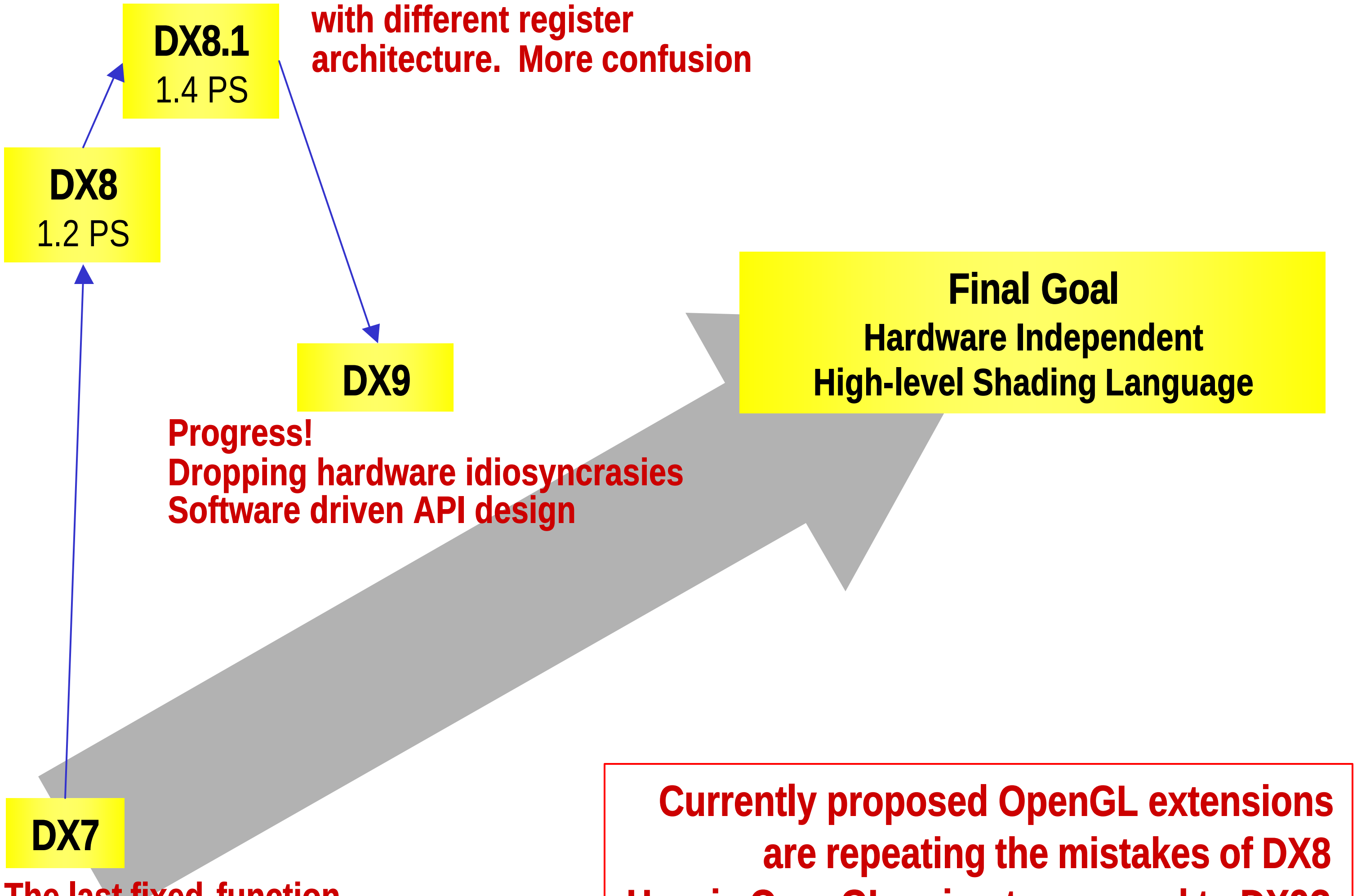
Does OpenGL Really Need Change?

Yes! Seize the moment!

- **OpenGL 1.0 was finished in 1992**
  - It has served us well
- **Graphics Processors are becoming programmable**
  - A fundamental industry transition
- **Some original design assumptions are no longer valid**
  - “What’s fast” is different today than it was in 1991
- **System architecture has evolved significantly**
  - CPU, memory & system buses have evolved rapidly, memory is much, much cheaper
- **Over 230 OpenGL extensions have been defined**
  - Nvidia's extension documentation is 500+ pages, the OpenGL 1.3 spec is 284 pages
- **Direct3D has overtaken OpenGL with some advanced features**
  - Microsoft have taken some bold design decisions
- **Many ISVs are considering transitioning from OpenGL to D3D**
  - For advanced features and performance
- **The time is right to make some bold, forward looking changes**
  - Programmable hardware revolution is an ideal inflection point to move OpenGL forward

D3D and Programmability

Some API design mis-steps, but making good progress



Some limited programmability but hardware-centric API design makes problematic compiler target

Another hardware-centric API with different register architecture. More confusion

Final Goal
Hardware Independent
High-level Shading Language

Progress!
Dropping hardware idiosyncrasies
Software driven API design

The last fixed-function D3D API

Currently proposed OpenGL extensions are repeating the mistakes of DX8
How is OpenGL going to respond to DX9?

A Perspective on Open Standards

As a positive force for market development

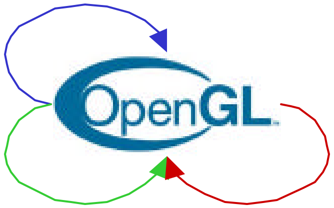
- **OpenGL 1.0 was a stroke of genius**
  - It set a long-term agenda for hardware vendors to strive for – pulling the industry forward

Primordial Soup
IRIS GL, PHIGS
Pex, GKS, Core

A Coherent Vision of the future

OpenGL

- **Today, OpenGL can be implemented on a chip**
  - ARB discussing which features from existing chips to standardize
  - Not a positive dynamic for a forward-looking industry standard



No Forward Progress!
IP, not customer needs, dominates many discussions

- **Need new long-term vision – Hardware Independent Shading Language!**
  - Defines the long-term agenda for the new generation of programmable hardware

OpenGL

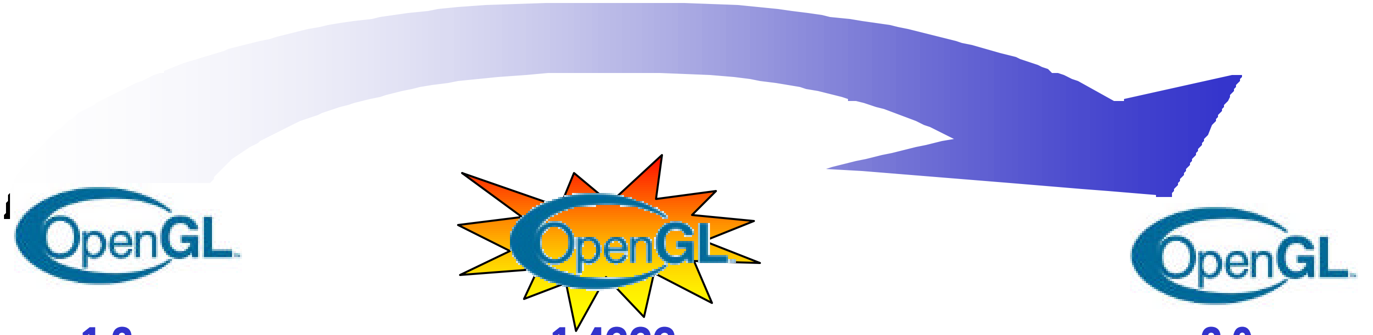
A coherent Vision of the future

High-level Programmability!

Hardware Independence is the Key

Programmability without stalling the ARB

- **We need to raise the level of debate**
- **Set out a future vision for the industry as OpenGL 1.0 did**



1.3
Just
Ratified

1.4???
Extended current chip
implementation discussions
Slow progress.
Functionality gap versus
D3D widens further.

2.0
Long terms goals create
hardware independent,
unifying direction

Holistic Approach to OpenGL 2.0

Solving more than "just" the programmability issue

Hardware independent high-level shading language

Tightly integrated with OpenGL

Simplify OpenGL

Replace complexity with programmability

Time Control

Generalized, unified solution for parallelism and timing estimation and control

Streamline Memory Management

Simple yet powerful mechanisms to enable application control of memory policies

OpenGL 2.0 Design Goals

A holistic approach for a new generation OpenGL

- **100% backward compatibility with OpenGL 1.3**
  - Include complete OpenGL 1.3 functionality
  - Enable OpenGL 2.0 programmability to be mixed with 1.3 functionality
- **Enable a smooth transition to a simplified OpenGL 2.0 subset**
  - A well-defined profile of forward looking functionality
- **Set an agenda for future hardware**
  - Don’t be limited to what hardware can do today
- **Generalized support for multi-pass rendering**
  - Define a new data buffer to hold generalized data for programmable processing
  - Either spatially or FIFO buffered data
- **Further optimize data movement between host and graphics subsystem**
  - A key performance enabler

Process for Creating OpenGL 2.0

A call for cooperation at the ARB

- **3Dlabs is willing to generate an OpenGL 2.0 specification proposal**
  - We are willing to commit the necessary intellectual bandwidth and resources
  - Regular reviews of the specification at the ARB and in the wider industry
- **We need to cooperate with ARB members and interested third parties**
  - To make this work genuinely vendor independent and be a positive direction for OpenGL
  - To deploy in a meaningful timeframe
- **3Dlabs is discussing this proposal without an NDA**
  - Instead, a simple agreement that states any IP disclosed by any party in these discussions will be made available royalty-free if included in the OpenGL 2.0 specification

Academia
Basis in independent research creates a non-partisan foundation

Stanford wants to contribute technology and review progress

Hardware Vendors
All stand to benefit
The industry should unite!

Feedback so far has been uniformly positive

2.0

OpenGL-based ISVS
They want it, they need it

Very positive feedback from DCC, visualization and simulation ISVs

Expediting OpenGL 2.0 Deployment

We need much more than a spec

- **We need to agree how we can work together to implement and deploy**
  - Cooperative resources and funding to expedite timescales
- **Rapid reference software implementation**
  - Can ship initially as OpenGL 1.3 extensions
  - Prove detailed design before ratifying specification
- **Define and implement complete OpenGL 2.0 package**
  - Conformance tests, benchmarks, development tools, marketing support etc. etc..
- **Suggested timescales**
  - Circulate White Papers on main design aspects – 2H01
  - Initial detailed review – December ARB meeting
  - Reference implementation and specification refinement – 1H02
  - Final draft ready for ratification - Siggraph 2002
- **Need feedback from this meeting on interest and possible resources**
  - But first… more technical details…

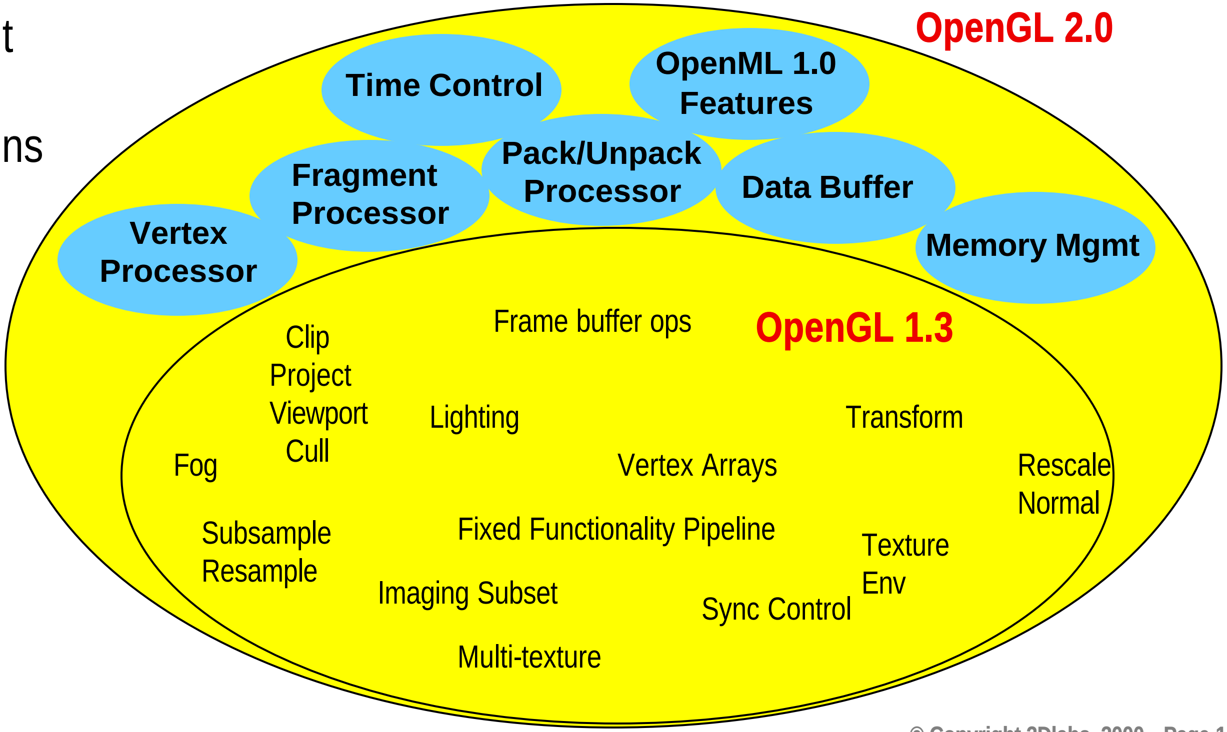
Future Directions

Technical Discussion
September 2001

Reinvigorating OpenGL

With 100% backwards compatibility for existing apps

- **OpenGL 2.0 adds powerful new features to OpenGL 1.3**
  - OpenGL 2.0 includes both new functionality and OpenGL 1.3 functionality
- **New OpenGL 2.0 functionality can be used incrementally by apps**
  - Vertex programmability
  - Fragment programmability
  - Pack/unpack programmability
  - Time control
  - Memory management
  - Data buffer
  - OpenML 1.0 extensions



OpenGL 2.0

Time Control

OpenML 1.0 Features

Fragment Processor

Pack/Unpack Processor

Data Buffer

Vertex Processor

Memory Mgmt

OpenGL 1.3

Frame buffer ops

Clip
Project
Viewport
Cull

Lighting

Transform

Fog

Vertex Arrays

Rescale Normal

Subsample
Resample

Fixed Functionality Pipeline

Texture Env

Imaging Subset

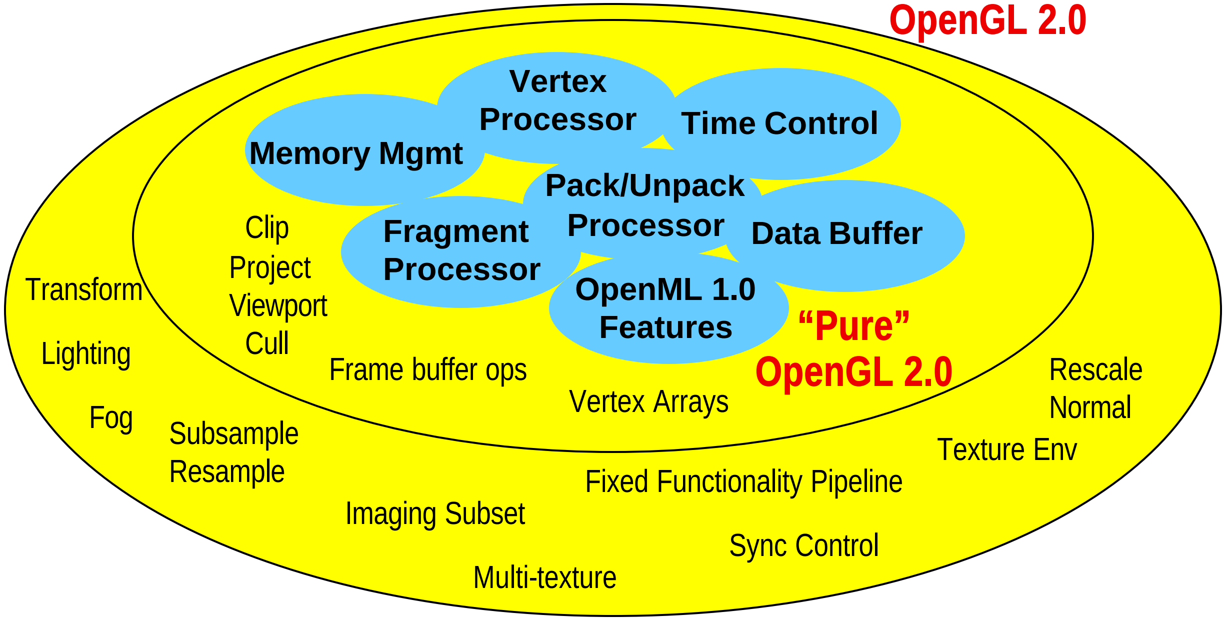
Sync Control

Multi-texture

“Pure” OpenGL 2.0

Defining the direction for future versions of OpenGL

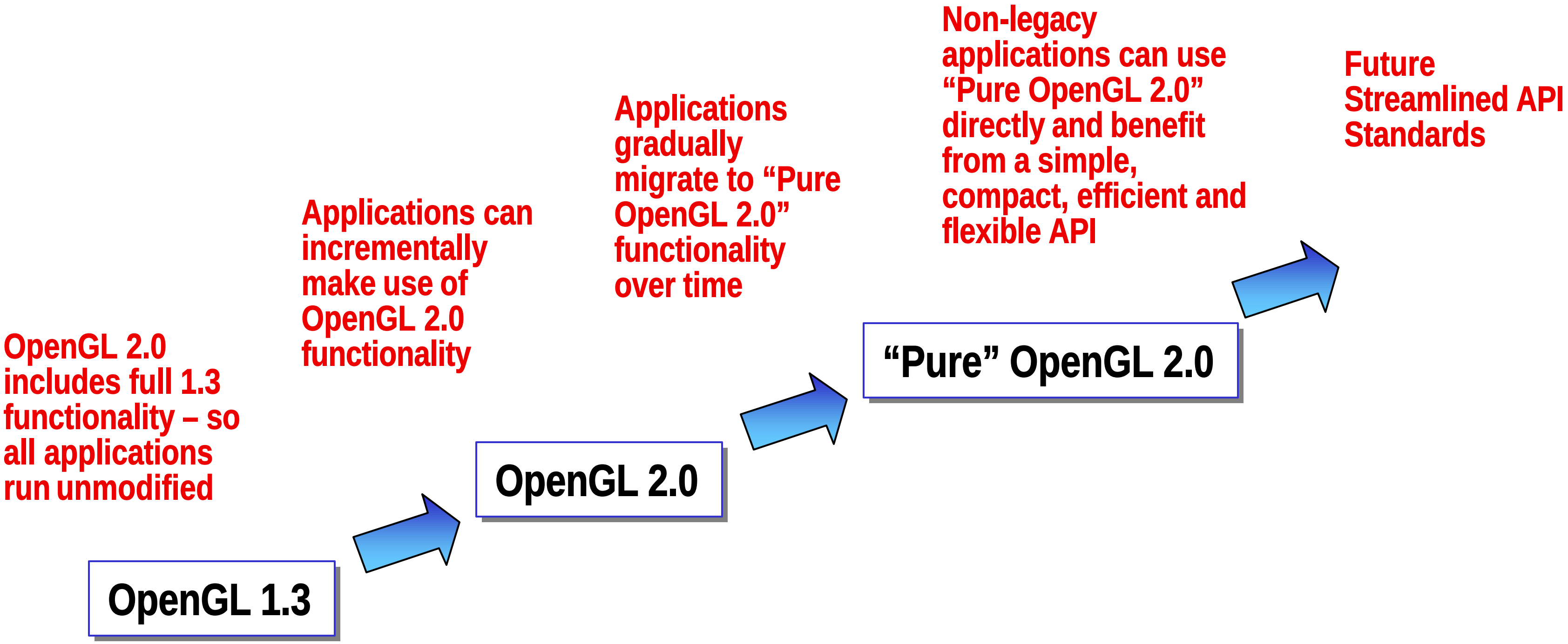
- **Replaces complexity with programmability**
  - Transform and lighting
  - Texture application
  - Pack/unpack pixels
  - Lots and lots of extensions
- **Cleaner API, smaller footprint, less complexity**
  - Easier and cheaper to implement



Smooth Transition Path

For existing and non-legacy OpenGL applications

- **Pure OpenGL 2.0 is the path forward**
  - Reduces implementation burden for markets where legacy support is not required
  - Replaces complexity with programmability
  - Drastically reduces the need for extensions
  - Legacy functionality could be deprecated once apps have migrated



OpenGL 1.3

OpenGL 2.0 includes full 1.3 functionality – so all applications run unmodified

OpenGL 2.0

Applications can incrementally make use of OpenGL 2.0 functionality

Applications gradually migrate to “Pure OpenGL 2.0” functionality over time

“Pure” OpenGL 2.0

Non-legacy applications can use “Pure OpenGL 2.0” directly and benefit from a simple, compact, efficient and flexible API

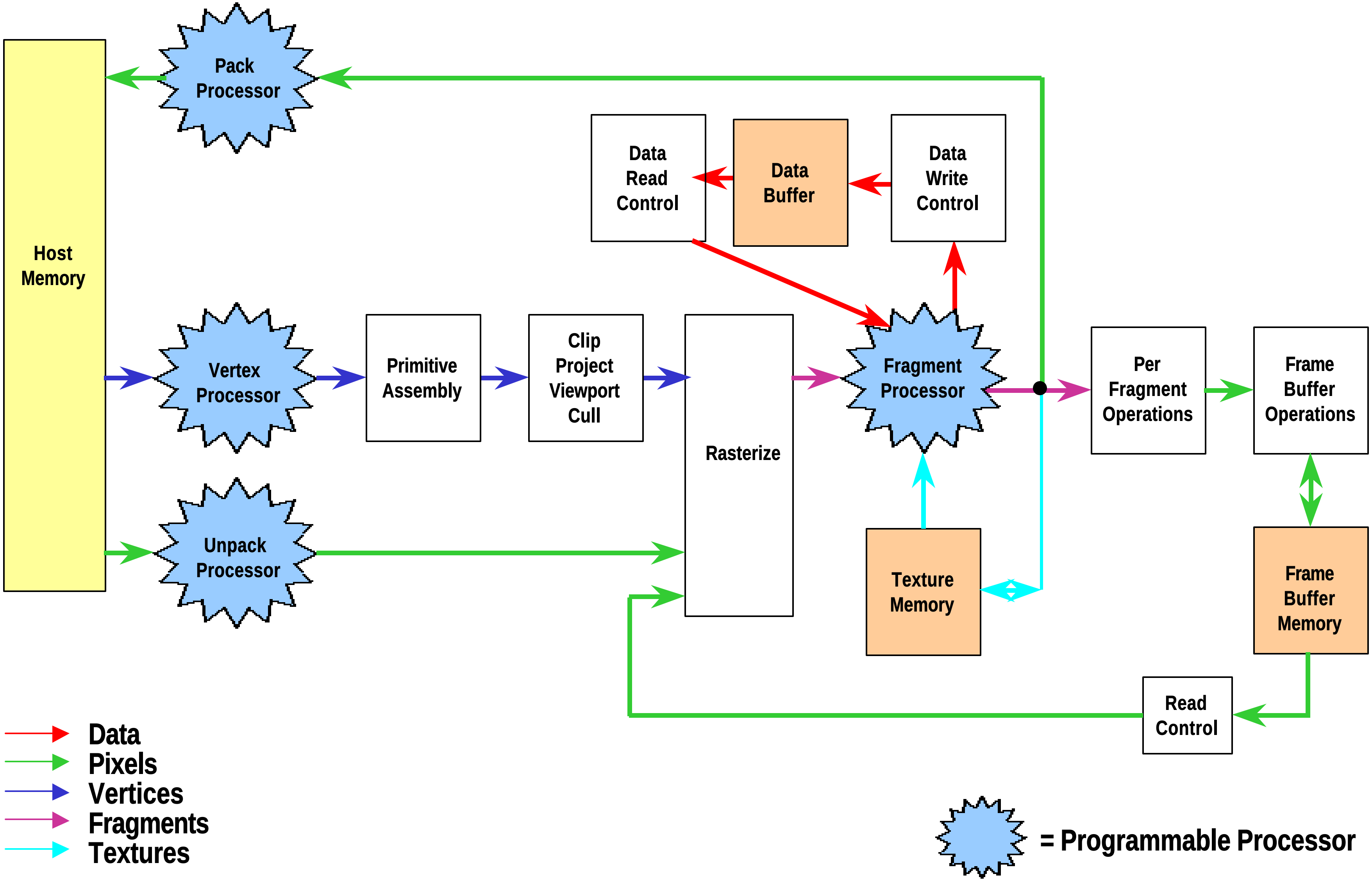
Future Streamlined API Standards

Programmability Philosophy

Add flexibility where it is most needed

- **OpenGL is a good design, just inflexible**
- **Add programmability where innovation and customization most needed**
  - Look at where most additions/changes have been made to OpenGL
- **Keep fixed functionality for areas where flexibility is less compelling**
  - Frustum clipping, backface culling, viewport mapping
  - Fixed functionality can still be disabled or set to identity mapping for full programmability
- **Keep fixed functionality for areas where hardware implementation is cheap, programmability is complex**
  - Frame buffer operations that involve read/modify/write operations
- **Programmability replaces fixed function state machines in several areas**
  - Vertex Processor
  - Fragment Processor
  - Pixel Pack
  - Pixel Unpack
- **Make programmability HARDWARE INDEPENDENT**

OpenGL 2.0 Data Path Diagram



Host Memory

Pack Processor

Vertex Processor

Unpack Processor

Primitive Assembly

Clip Project Viewport Cull

Rasterize

Data Read Control

Data Buffer

Data Write Control

Fragment Processor

Texture Memory

Per Fragment Operations

Frame Buffer Operations

Frame Buffer Memory

Read Control

Data
Pixels
Vertices
Fragments
Textures

= Programmable Processor

Vertex Processor Capabilities

Lighting, material and geometry flexibility

- **Vertex programs replace the following parts of the OpenGL pipeline:**
  - Transform
  - Normalization
  - Lighting
  - Texture coordinate generation
  - Fog
- **The vertex shader does NOT replace:**
  - Perspective projection and viewport mapping
  - Clipping
  - Backface culling
  - Primitive assembly
  - Two sided lighting selection
  - Polymode processing
  - Polygon offset
  - User clipping support

Fragment Processor Capabilities

Texture access, interpolator & pixel operation flexibility

- **Fragment programs replace the following parts of the OpenGL pipeline:**
  - Smooth shading
  - Texture access
  - Texture application
  - Alpha test
  - Pixel transfer including ARB Image extensions
- **The fragment shader does NOT replace:**
  - Pixel ownership test
  - Depth test
  - Stencil test
  - Alpha blending
  - Logical ops
  - Dithering
  - Plane masking

Pack/Unpack Processor

Replaces complexity of pixel storage operations

- **Pack/unpack processor capabilities**
  - Flexible pixel formatting as data is streamed to/from host memory
  - Provides access to flexibility through programmability
  - Produces a coherent stream of pixel data
- **Pack/unpack processing replaces**
  - Complexity of pixel formats/types
- **Eliminates need for pixel format extensions**
  - EXT_cmyka
  - SGIX_ycrcb
  - EXT_422_pixels
  - OML_subsample
  - OML_resample
  - NV_packed_depth_stencil
  - And variations of all the above

Shading Language for OpenGL 2.0

Hardware independent

- **Integrated intimately with OpenGL 1.3**
  - Existing state machine is augmented with programmable units
- **Enables incremental replacement of OpenGL 1.3 fixed functionality**
  - E.g. make simple lighting changes without having to rewrite parameter management
- **C-based - with comprehensive vector and matrix types**
  - Also integrates some RenderMan features
- **Virtualizes pipeline resources**
  - Programmers, for the most part, needn’t be concerned with resource management
- **Same language for Vertex Program and Fragment Programs**
  - Some specialized built-in functions and data qualifiers

Shading Language Structure

Types

- **float**
  - a single floating point scalar
- **vec2, vec3, vec4**
  - floating point vector
- **mat2, mat3, mat4**
  - floating point square matrix
- **int**
  - 16 bit integer for loops and array index
- **texref**
  - synonymous with texture stages

Shading Language Structure

Qualifiers

- **const**
  - variables is a constant and can only be written during its declaration
- **attribute**
  - vertex input data to the Vertex Shader
- **uniform**
  - a constant provided via the API
- **varying**
  - an interpolated value
  - output for Vertex Shader
  - input for Fragment Shader
- **Uniform and varying variables can be aggregated into arrays**
  - [ ] to access element
- **Layout of attribute, uniform and varying fixed by position in block { }**
  - as in structures, used to direct binding between API/vertex program and fragment program

Shading Language Structure

Expressions

- **constants**
- **variables**
- **scalar/vector/matrix operations as expected**
- **+, -, * and /**
- **built in functions**
- **user functions**
- **Component accessor for vectors and matrices**
- **Row and column accessors for matrices**

Shading Language Structure

Flow Control

- **if, if else**
- **for**
- **while**
- **break and continue supported**

Shading Language Structure

Functions

- **User Functions**
  - Arguments passed by reference
  - Modifiable arguments qualified with output keyword
  - Function overloading
  - May need to cast return to resolve ambiguities
  - One return value (can be any type)
  - Scope rules same as C
- **Built-in Functions work on scalars and vectors**
  - sin, cos, tan, asin, etc.
  - pow, log2, exp2, sqrt, inversesqrt
  - abs, sign, floor, ceil, fract, mod, min, max, clamp
  - mix, step, smoothstep
  - length, distance, normalize, dot, cross
  - element (Vertex Program only)
  - texture, lod, dPdx, dPdy, fwidth, kill, noise (Fragment Program only)

Example Fragment Program

Gouraud shaded, modulated, fogged fragment

```
// State
uniform
{
        vec3    fogColor;
};

// Interpolated parameters (input from vertex program)
varying
{
        vec2    texCoord;
        vec4    color;
        float   fog;
};

texref  texMap = 1;             // define OpenGL texture 'stage'

void main (void)
{

        vec4            col;
        float           f;

        col = color * vec4 texture(texMap, texCoord);
        f = clamp(fog, 0.0, 1.0);
        fragColor = mix(col, fogColor, f);      // standard frag color
}
```

Vertex Program Example

Dual directional light, texture and fog – 1 of 3

```
// State
uniform
{
           mat4       ModelViewMatrix;
           mat4       ModelViewProjectionMatrix;
           mat3       NormalMatrix;
           vec3       Material[5];
           float      MaterialSpecularExponent;
           vec3       Light0[7];
           vec3       Light1[7];
           float      FogStart, FogEnd;
};

// Vertex attributes
attribute
{
           vec4       inPosition;
           vec3       inNormal;
           vec4       inTexture;
};

// Output to Fragment Program (position is predefined).
varying
{
           vec2       texCoord;
           vec4       color;
           float      fog;
};

// Globals
vec3       ambientIntensity;
vec3       diffuseIntensity;
vec3       specularIntensity;
vec3       normal;
```

Vertex Program Example

Dual directional light, texture and fog – 2 of 3

```
void main (void)
{
        float   eyeZ;

        // Transform vertex to clip space
        position = ModelViewProjectionMatrix * inPosition;

        normal = NormalMatrix * inNormal;

        // Clear the light intensity accumulators
        ambientIntensity = vec3 (0);
        diffuseIntensity = vec3 (0);
        specularIntensity = vec3 (0);

        DirectionalLight (Light0, MaterialSpecularExponent);
        DirectionalLight (Light1, MaterialSpecularExponent);
        color = Material (Material);

        // Just copy through for simplicity.
        texCoord = vec2 (inTexture.s, inTexture.t);

        // Simple linear fog.  Note 3_ notation selects row 3 of matrix
        eyeZ = dot (ModelViewMatrix.3_, inPosition);
        fog = (FogEnd - abs (eyeZ)) * (FogEnd-FogStart);
}
```

Vertex Program Example

Dual directional light, texture and fog – 3 of 3

```
void DirectionalLight (vec3 lightInfo[], float specularExponent)
{
        float   normalDotVP;
        float   normalDotHalfVector;
        float   powFactor;

        normalDotVP = min(0, dot (normal, lightInfo[ePosition]));
        normalDotHalfVector = min(0, dot (normal, lightInfo[eHalfVector]));

        if (normalDotVP == 0.0)
                powFactor = 0.0;
        else
                powFactor = pow(normalDotHalfVector, specularExponent);

        ambientIntensity += lightInfo[eAmbientIntensity];
        diffuseIntensity += lightInfo[eDiffuseIntensity] * normalDotVP;
        specularIntensity += lightInfo[eSpecularIntensity] * powFactor;
}

vec4 Material (vec3 mat[])
{
        vec3 color;

        color = mat[eEmissive] + mat[eAmbient] * SceneAmbient +
                                    ambientIntensity * mat[eAmbient] +
                                    diffuseIntensity * mat[eDiffuse] +
                                    specularIntensity * mat[eSpecular];

        return vec4(color.red, color.green, color.blue, mat[eDiffuseAlpha]);
}
```

Comparing OpenGL 2.0

To the Stanford Shading Language

- **RenderMan provides the whole graphics pipeline (and a lot more)**
  - Has no connection to OpenGL
  - No compromises for real time or direct hardware implementation
- **The Stanford Shading Language is at a lower level than RenderMan**
  - More geared for hardware support
  - Still sits on top of OpenGL and abstracts lights and surfaces which are defined separately
- **OpenGL 2.0 language is lower level still - integrated into OpenGL itself**
  - Provides alternative methods to the state controlled pipeline
  - This has some advantages and disadvantages…

Stanford Shading Language

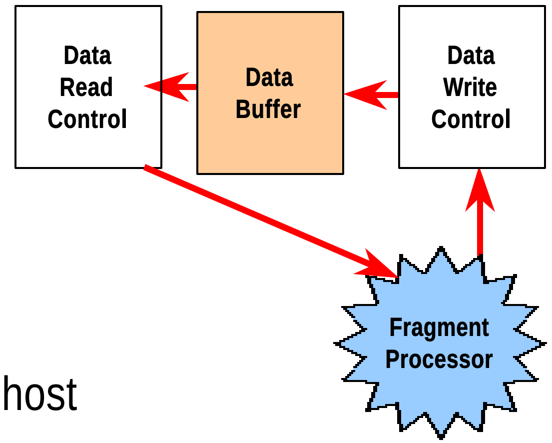
Pros and Cons

- **Stanford has a higher level of abstraction than OpenGL 2.0**
  - Easier programs where the abstraction matches the problem domain
- **OpenGL 2.0 retains the level of abstraction of the current OpenGL**
  - Allows intimate blending of old and new functionality
  - Incremental adoption of OpenGL 2.0 features
  - Appropriate abstraction level for video and image processing
  - Can layer higher levels of abstractions on top of a programmable OpenGL
- **Stanford compilers detect computational frequencies**
  - Enabled by higher abstraction level that defines the total graphics operation
  - Compiler can detects what runs at the primitive group, primitive, vertex & fragment levels
  - Compiler can generate code for CPU, vertex processor and fragment processor
- **OpenGL 2.0 exposes vertex, fragment and pixel operations**
  - Incremental integration into existing OpenGL state machine
  - glBegin and glEnd are still available as are display lists and other core OpenGL features
  - Can still use the standard OpenGL T&L operations with a custom fragment shader
- **This language proposal is “OpenGL-like” and adds programmability without re-designing the whole OpenGL infrastructure**

The Data Buffer

A new OpenGL buffer

- **Simple, generalized concept to enhance programmability**
  - Obviates the need for specialized future functionality extensions
- **Can hold any data**
  - Enables multi-pass fragment processor programs
- **Supports stream processing**
  - Not random RW
- **Flexible addressing**
  - Spatially buffered data
  - FIFO buffered data
- **Usage examples**
  - Multiple outputs from fragment processor
  - Intermediate results
  - Multi-spectral processing
  - Sweeney style color/normal buffer
  - Can render floating point images and read-back to the host for back-end rendering acceleration



Generalized Time Control

Improve parallelism and control of operation timing

- **OpenGL traditionally worries about how and where - not when**
  - Need better mechanisms for knowing and controlling when things happen
- **Allow work to continue while waiting for something else to finish**
  - For example, clear depth buffer while waiting for a swap
- **OpenGL needs a generalized and unified synchronization mechanism**
  - Eliminates the need for each extension to invent its own mechanism
  - Solves short term problems – such as parallel image download
  - Enables more sophisticated use of timing control in the future
- **Enable application to dictate timing policy**
  - Controlling when things are going to happen
- **Allow parallel execution of longer OpenGL operations**
  - Long operation “runs in the cracks” with respect to main rendering
- **Provide better replacements for flush and finish**

Generalized Time Control

Underlying Mechanisms

- **Synchronization data type**
  - Useful across many parts of OpenGL
- **Stronger flush**
  - To guarantee host and graphics parallelism
- **Fences**
  - Better alternative to finish
  - Synchronize between host and graphics
  - Synchronize between two graphics streams
- **Background context selection**
  - Schedule long task to a parallel background stream
  - Allow priority for isochronous operations
- **Asynchronous Operations**
  - Enables asynchronous data binding of long commands
  - Frees thread to do other things while the asynchronous operation is underway
  - Allows other rendering to occur while waiting to start asynchronous operation
- **Eliminates OpenML extensions and existing fence extensions**
  - SyncControl, Async, AsyncPixels
  - Nvidia’s Fence extensions

Holistic Memory Management

Current OpenGL memory management is a black box

- **Everything done automatically by OpenGL**
  - Application does not know when an operation will happen
  - Application does not know how long an operation takes
  - Application does not know how much backing store is allocated

Application does not have control over:
  - Where objects are stored (textures, display lists, vertex arrays)
  - When objects are copied, moved, deleted, or packed (defragmented)
  - Virtualization of memory resources

OpenGL has very limited 'query for space' functionality
  - Proxy textures only

Holistic Memory Management

Application can get control, if it wants

- **Application sets a memory management policy for each object**
  - OpenGL should not dictate policy
  - Application should be able to adopt and dynamically change a policy
- **Unifying mechanisms for all objects**
  - Textures, display lists, vertex arrays, images
  - Policies, priorities, queries, timing estimates
- **No distinction between texture, display list & vertex array memory etc.**
  - Does not prevent OpenGL implementations from having these object specific pools
- **Application can still let OpenGL manage memory**
  - Just another policy

Holistic Memory Management

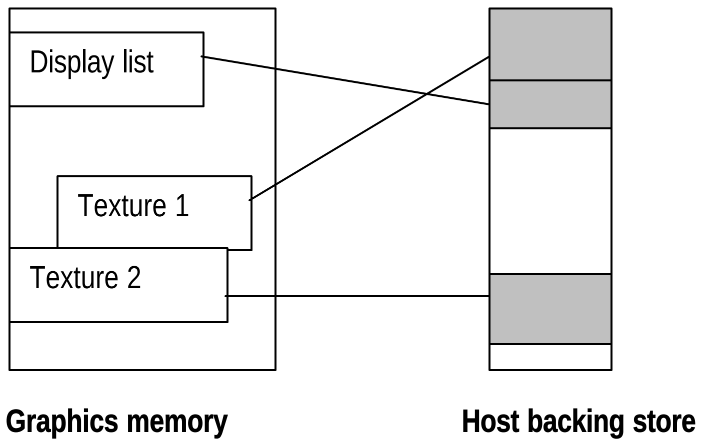
Underlying Mechanisms

- **Query for Space and Make Space functions**
  - Ask if there is room for an object or objects with a certain policy
  - Make room for an object or objects with a certain policy
- **Set priorities for all objects, not just textures**
  - Used when querying for space and making space
- **Time estimates**
  - How long does a memory management operation take
- **Set and unset a policy**
  - Two currently defined policies – more can be added
  - Pinned and OpenGL default
- **Pinned policy – application gets control**
  - OpenGL will not move, copy, pack or delete an object once data loaded
  - The object is pinned down in high performance graphics memory
  - Application is responsible to store or delete objects, or to initiate a packing operation

OpenGL 1.3 Memory Management

Some outstanding issues

- **OpenGL memory layout today**

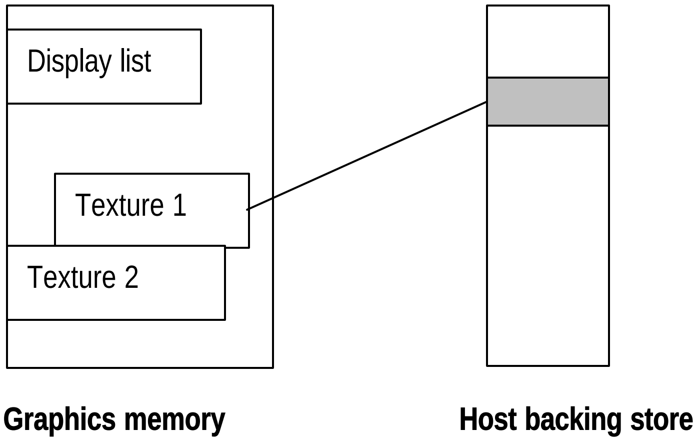


- No control over when or where objects are stored, or when deleted
- No control if, and when, object gets copied or memory defragmented
- Typically allocates backing store

OpenGL 2.0 Memory Management

Issues fixed

- **New OpenGL memory layout**



- Texture 1 GL_POLICY_DEFAULT
- Display list and Texture 2 have GL_POLICY_PINNED set
- Note, backing store not necessarily allocated for pinned objects

Vertex Data Host Performance

Minimize data movement from host to graphics

- **Both static and dynamic vertex arrays**
  - Dynamic – OpenGL today
  - Compiled vertex arrays
  - Vertex array objects
- **Enhance vertex arrays**
  - Vertex array memory mapping
  - Index arrays for each data type
- **Asynchronous vertex array data binding**
  - Relax current OpenGL constraints
- **Direct access to graphics memory**
  - Image buffer memory
  - Vertex array memory
- **More efficient primitive types**
  - Grid primitives
- **De-emphasize immediate mode glBegin/glEnd**

ARB / Khronos Convergence

OpenML 1.0 Requirements versus OpenGL 2.0 proposal

- **OpenGL imaging subset**
  - Implemented in the fragment processor
- **OML_subsample/resample extensions**
  - Implemented in a pack/unpack processor
- **OML_interlace extension**
  - Implemented in a pack/unpack processor
- **OML_sync_control extension**
  - Implemented using the “time control” mechanisms
- **SGIS_texture_color_mask**
  - Need to add to OpenGL 2.0

An increasingly common set of requirements – significantly met by OpenGL 2.0. Should we consider tighter integration of Khronos and ARB activities?

OpenML 1.1 Requirements

Mapped against OpenGL 2.0 proposal

- **Extended precision and greater dynamic range for colors**
  - Fragment processor, data buffer help achieve what's needed
- **Render to texture**
  - Could be defined as part of OpenGL 2.0
- **Memory & texture management improvements**
  - Combination of "memory management" mechanisms and fragment processor
- **Non power of 2 textures**
  - Still needs a resolution
- **Pixel shaders and vertex shaders**
  - Provided by fragment processor and vertex processor
- **Async extension**
  - Satisfied by "time control" mechanisms
- **Texture filtering for interlaced video**
  - Customized filters implemented in fragment processor
- **Support for integrated video and graphics**
  - Enhanced with programmable flexibility

Summary

The OpenGL 2.0 Initiative

- **We believe that this is the right time to define OpenGL 2.0**
  - Programmability marks a fundamental industry shift – requiring a new API approach
  - Programmability is an opportunity to reduce API complexity
- **3Dlabs is proposing a holistic approach to OpenGL 2.0**
  - Hardware independent shading language, time control, memory management
  - “Pure” OpenGL 2.0 provide a smooth transition path to a simple, flexible, powerful API
- **This needs to be a cooperative project**
  - 3Dlabs volunteering to develop the specification – with full industry participation
  - Need wide cooperation for implementation and infrastructure development
- **An opportunity to let the ARB move forward again**
  - Developing standards that set a direction for the industry
  - Enabling closer cooperation with Khronos

- **Let’s work together to ensure OpenGL continues to meet the needs of the graphics industry in the 21st century**

Outstanding Issues

We need your feedback

- **How to resource this initiative**
  - Who is interested to help?
- **The correct balance for backward compatibility**
  - What is the correct balance between full and “pure” OpenGL 2.0
- **Technical questions**
  - Tessellation - still an area of research?
    - Generalize a single vertex processor or add another unit?
  - Structures do not currently exist in OpenGL
    - What were the issues?
  - PINNED_ALWAYS is a potentially dangerous policy
    - But could it be needed in applications such as the embedded space
  - Language bindings
    - Is C sufficient?
  - Handles vs. Ids
    - Pros and cons