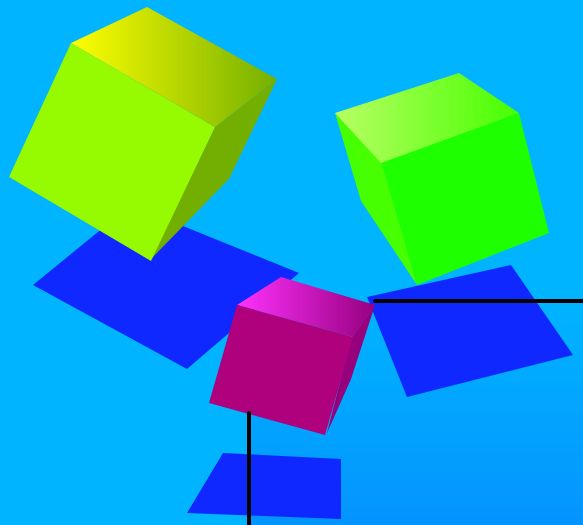
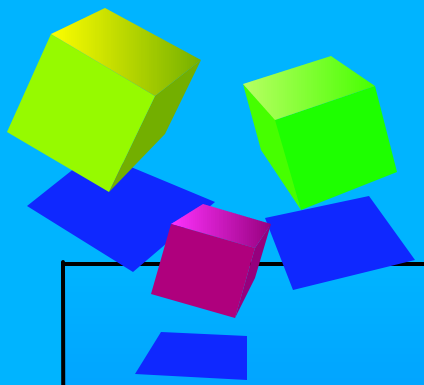


Programmable Hardware #1: Basic Hardware Capabilities

Jason Dale

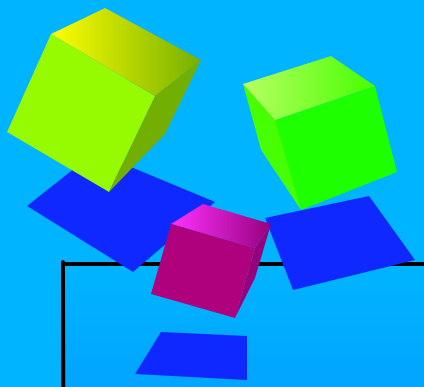
CS395T - Realtime Graphics

02/11/2003



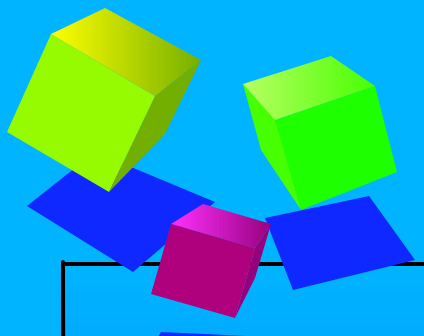
Overview

- My Perspective
- Review Paper 1: "Parallel Computers for Graphics Applications"
  - Goals
  - Chap - The predecessor to Flap
  - Flap
  - C4 extentions
- Review Paper 2: "A User-Programmable Vertex Engine"
  - ISA / Datatypes / Low-level programming
  - Hardware (briefly)
  - High-level programming
- Discussion and Thoughts on the Future



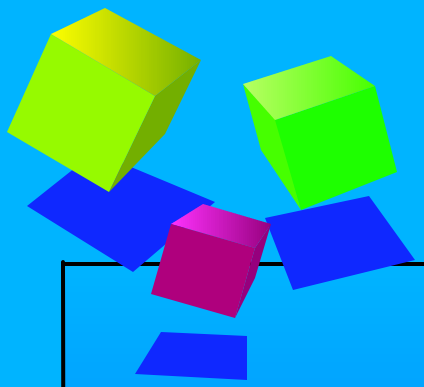
My Perspective

- Got the feeling that many people in the class are graphics whizzes
- My background is with "General Purpose" CPUs (Servers)
- Hardware-centric skew
  - Used to do logic design
  - Currenly do processor performance analysis
    - Benchmarks
    - Pipeline stages
    - CPI stacks
- Feel free to interrupt
  - For questions
  - To share interesting factoids
  - To correct - 3rd day with OpenGL ☺



Paper 1

- "Parallel Computers for Graphics Applications"
  - Adam Levinthal, Pat Hanrahan, Mike Paquette, Jim Lawson
  - Authors from Pixar - that's interesting
    - Typically think of "Toy Story" 3D animation
    - Work evolved from 2d film printer instead
  - Published in 1987
    - "Wow-factor" drops by $2^{(2003-1987)}$
    - Comparison Hardware: VAX 11/780 minicomputer
    - Flap is 15"x18" board (several chips running at 10 MHz)
    - Chap/Flap seem flexible -can see a modern GPU in there
- Have a quick overview of Chap
- Talk About Flap
  - Hardware and Addressing
  - C4 Data Structures and Programming Language



Chap overview

- ***CHA***nnel ***P***rocessor (an actual Pixar product)
- Designed for Back-end work (pixel operations)
  - 1024x1024 pixel 2D image processing
  - Image blending / compositing
  - Bluescreening
  - Edge filtering
  - Rotation/perspective transformation
  - Color space transformation / color correction
- 4-way SIMD
  - 4 color components - RGBA
  - 16 bit fixed-point is the fundimental data type
- 16 bit pre/post-increment addressing modes
- Uses C4 High-level Language

Flap overview

- ***FL***oating-point ***A***rray ***P***rocessor
- Designed for front-end work (points, normals, etc)
  - 3D transformations/clipping
  - Shading
  - Evaluation of cubic polynomials by finite difference
  - Geometric operations on meshes and quadrilateral
- Includes:
  - Integer processor (runs program, dictates control flow)
  - 32 bit single precision, 4way SIMD array (execution engine)
  - Interface busses
    - Sysbus (host interface)
    - Gbus (external high-speed memory and renderer interface)
  - 4 dedicated RAM banks and rotate logic
- More addressing modes than Chap (mul/div in agen)
- Also uses C4 Programming Language

Flap hardware

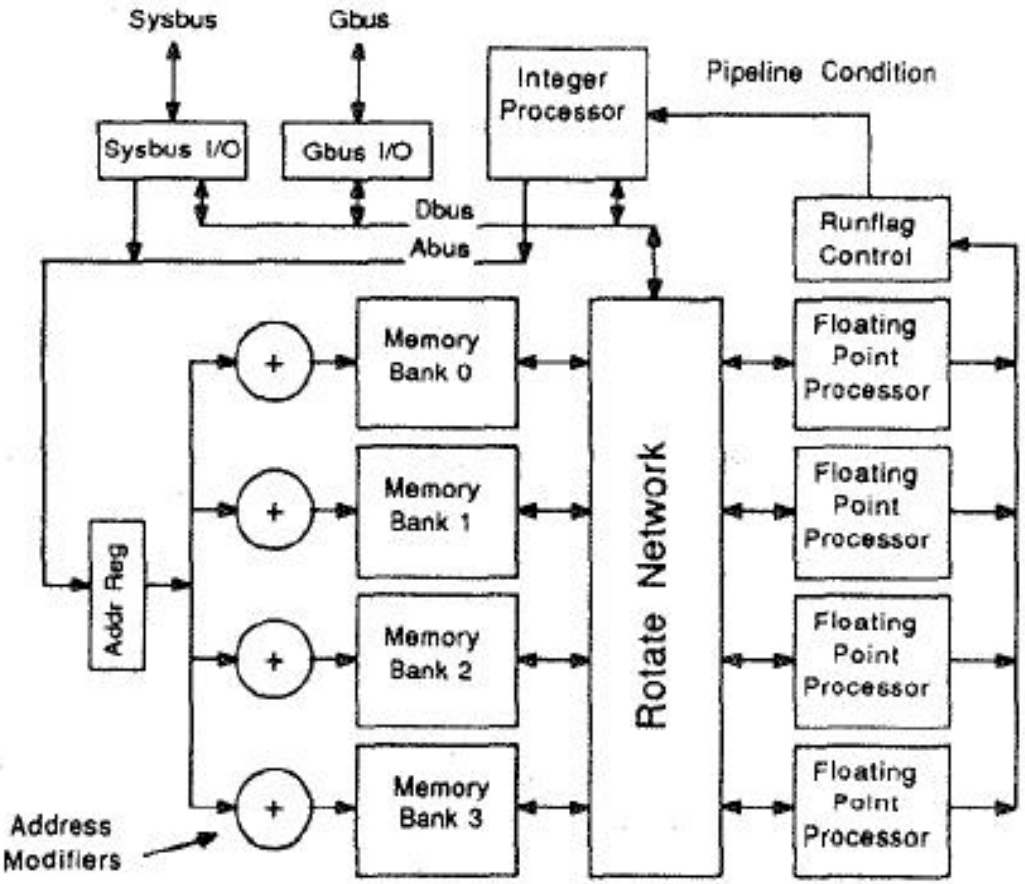
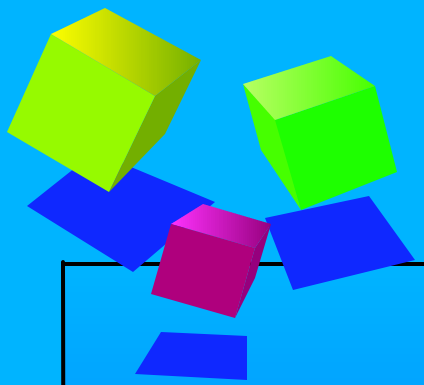
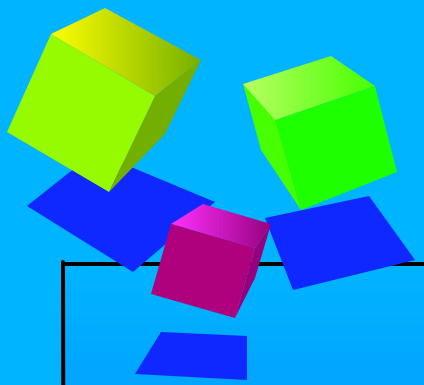


FIGURE 1: BASIC FLAP ARCHITECTURE



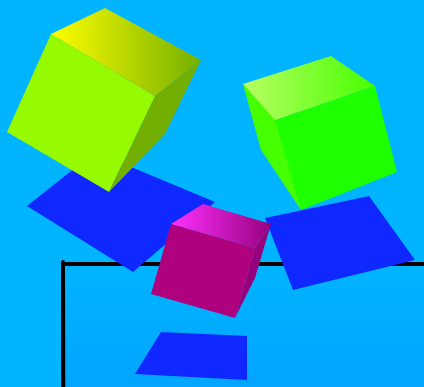
Flap low-level programming

- C4 Language for low-level parallelism
  - Superset of C - meshes with their C/UNIX development environment
  - Features to extract SIMD parallelism
    - Parallel Data Types
    - Individual Processor Control
      - Hardware not explained in great detail
      - Looks somewhat like predication (patented)
- Access types for scalar and vector data
  - word - 32 bit single precision
  - lrow - think it means "line row" / "logical row"
  - trow - think it means "tesselated row"
  - tcol - think it means "tesselated column"
- Tesselated memory design not clear



C4 Data Types

- Added "parallel" type modifier to C
  - Allows any data type to be replicated 4x (32bit only?)
  - Equivalent to a vector (or a row in a 4x4 matrix)
  - Typecasting defined (replicate ints to parallel ints)
  - Similiar to SIMD data types on other systems
  - Alignment rules weren't clear (pointers must be aligned?)
  - Uses a "trow" memory access
- Added "component" type modifier to C
  - Equivalent to a column in a 4x4 matrix
  - Uses a "tcol" memory access
  - Typecasting rules weren't clear



C and C4 statements

C4	C

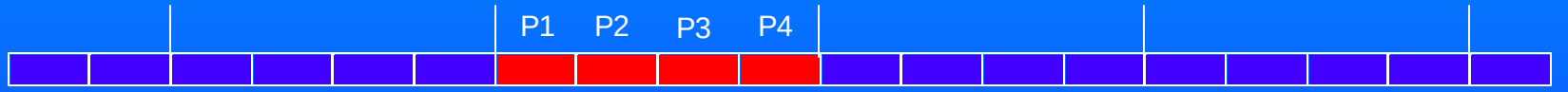
`parallel int x, y[2];`	`typedef struct par_int{`
	`  int chan[4];`
	`};`
	`par_int x, y[2];`

`x = y[1].chan[2];`	`x.chan[0] = y[1].chan[2];`
	`x.chan[1] = y[1].chan[2];`
	`x.chan[2] = y[1].chan[2];`
	`x.chan[3] = y[1].chan[2];`

Flap Access Types

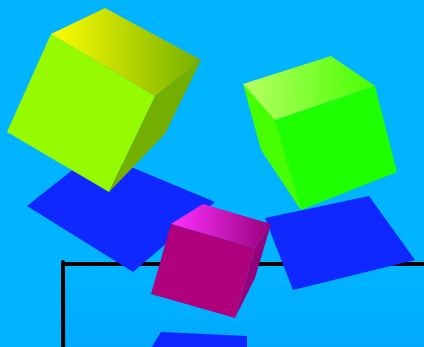
- Vector Access Types - calculated from a pointer
  - tword: single 32 bit value - replicated to all subprocessors in array

  - lrow: 4 consecutive words directly mapped to subprocessors



  - trow: 4 consecutive aligned words rotated to subprocessors

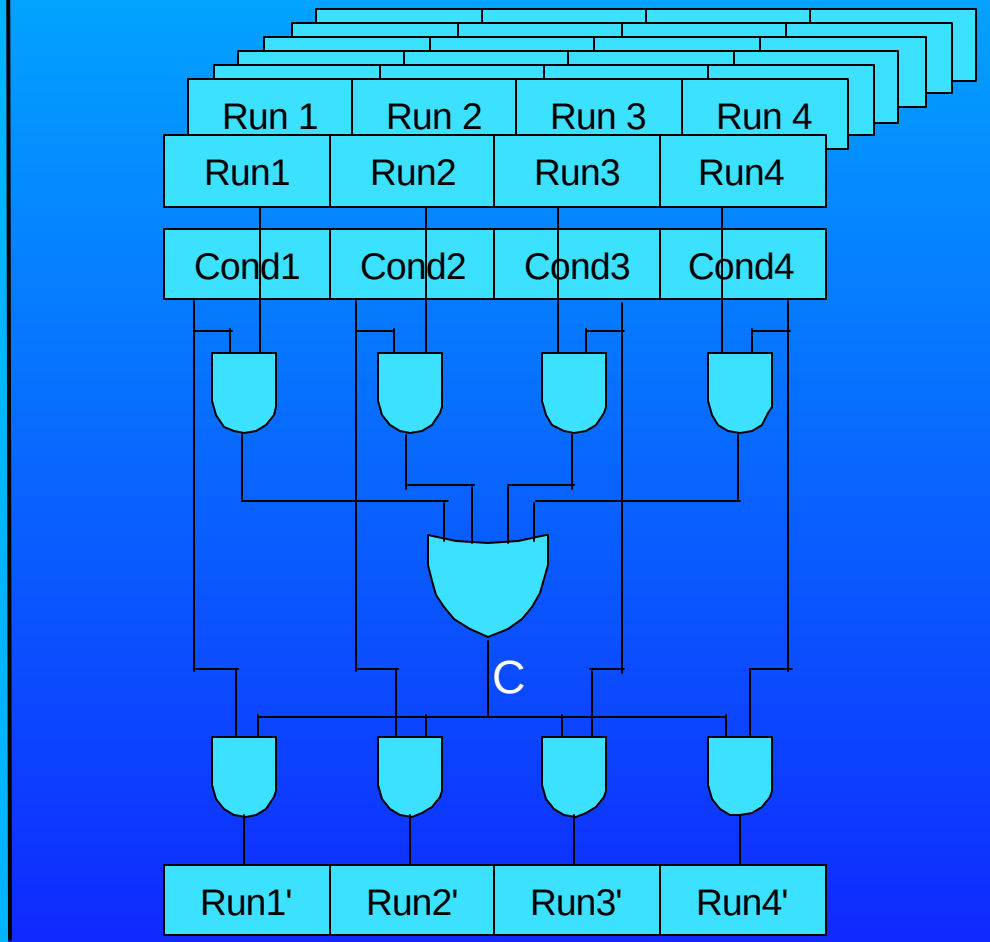
  - tcol: 4 words with stride of 4 words (component access)



Flap SIMD Control

- Code should be primarily SIMD (matrices, dot, cross, etc)
- But - need to control individual "subprocessors" sometimes
- Solution - "Runflag" control register and stack
  - Sub-processors can be individually enabled / disabled
  - One status and one run bit per subprocessor, global cond
  - Runflag Stack enables nested control flow
    - Push - Used when entering a do-while loop
    - Pop - Used at end of "if" statement
    - While - evaluates conditional and pops if false
    - If/else
      - Combination of push / evaluate / eval complement / pop
      - Both executed if cond / !cond on any running subprocessor
      - See logic on next page and the example
    - Set - Replace top of Runflag stack with an immediate value
    - Force - Force the Run bits for the current instruction only

Runflag Operation



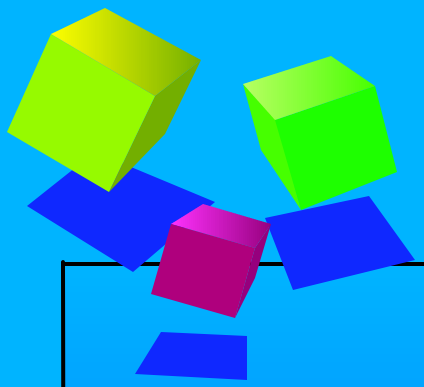
Runflag Stack

Registers

Conditional Evaluation

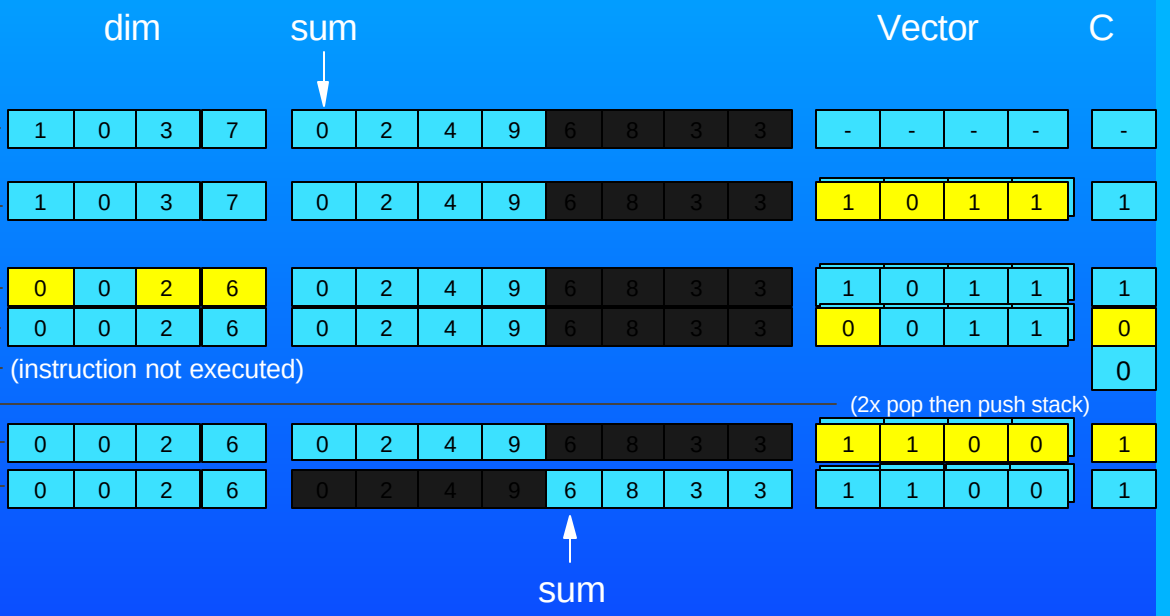
New Runflag

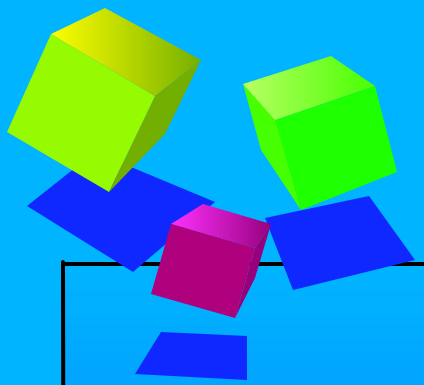
only updated if "C" is true



C4 Example Code

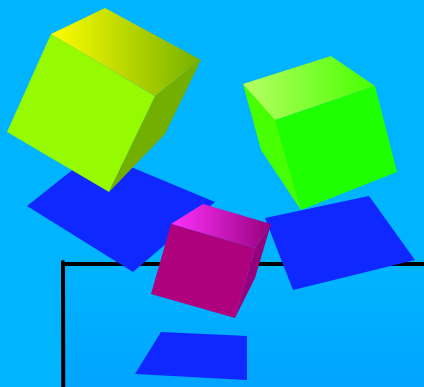
```
TestCase(Dim, Sum)
    parallel int dim;
    parallel int * sum;
{
    if (dim > 0)
    {
        dim = dim - 1;
        if(all(dim > 0))
            sum++;
    }
    if(any(dim == 0))
        *sum++;

    return (sum) ;
}
```





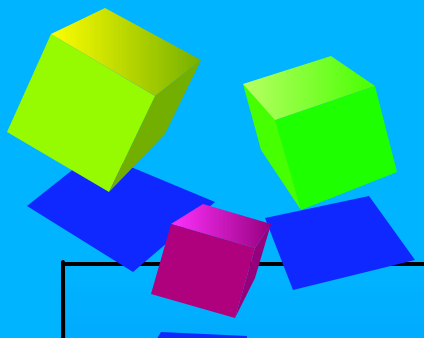
Runflag Control

- Runflag control is similiar to predication:
  - Integer Processor might execute both sides of a branch
  - Predication - typically enables/disables instruction writeback
  - Runflag - selectively enable/disable processor execution
- But unlike predication:
  - Can skip code blocks if condition is false on all subprocessors
    - More complex - merged predicate and branch mechanism
    - Efficient when skipped block has many instructions
  - Targeting something slightly different than modern predication
    - Isn't meant to eliminate control flow like most predication
    - Modern predication is good for long pipes / high frequency



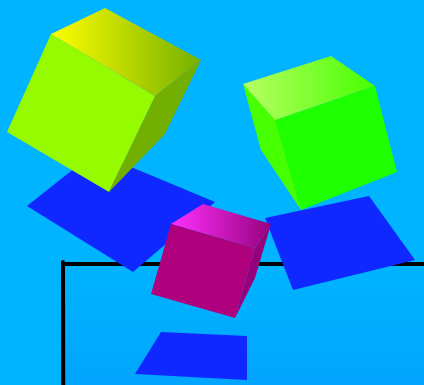
Questions

Questions on the first paper?



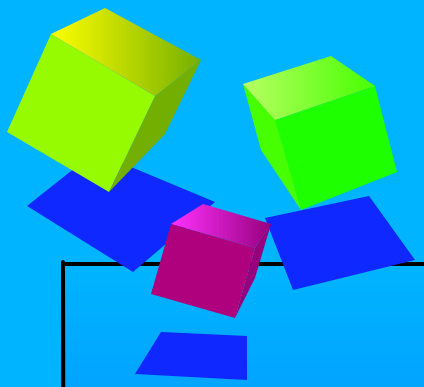
Paper 2

- "A User-Programmable Vertex Engine"
  - Erik Lindholm, Mark Kilgard, Henry Moreton
  - Published in 2001
    - Modern single chip GPU design (NVIDIA GeForce 3)
    - .18u, ~550pins, 700-960 Gops/s, 6-8 GB/s mem, 200+ MHz
  - Paper Focus is Vertex Shader (not Fragment/Pixel processor)
  - Presents a flexible / powerful / easy to program design
    - Originally evolved from fixed function pipeline
    - Exploits several forms of parallelism
      - 4x SIMD, single precision IEEE format (non IEEE modes)
      - Chip Multiprocessing (? not 100% clear on this)
        - Vertex processor (or processors?)
        - Multiple Pixel Shaders (Processor or state machine?)
      - Multithreading (transparent)
- Discuss datatypes and architecture
- Discuss hardware (briefly)
- Discuss programming model / API



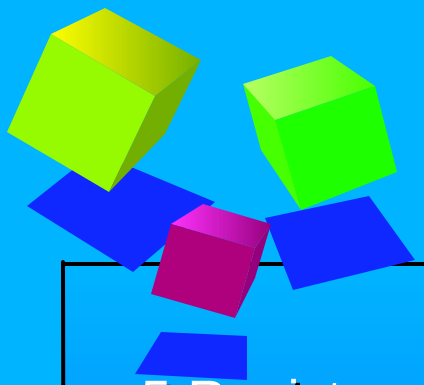
Motivations for design

- Programmability
  - Quickly evolving API needs programmable hardware
  - Programmable hardware lessens need for fixed API
- Ship-ability (dictates a lot of the architecture)
  - Design Time / Design Resources / Complexity
  - Platform independance / Standard API
  - Commodity Pricing (chip area / yield)
    - Do GPUs have an advantage over CPUs here?
  - Performance
- The motivation no-one ever talks about
  - What about marketing numbers?
  - CPU has GHz, GPU has GB/sec and Gops/s
  - Maybe with GPU, marketing numbers are not as evil



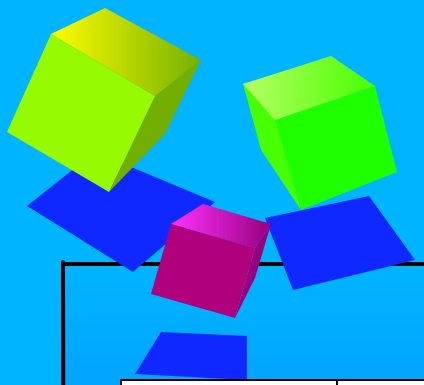
Programming Model

- Vertex is the most common element to operate on
  - High parallelism / low complexity with vertices
  - Triangle / polygon / other primitive could have been used
    - Better frustrum clipping, perspective divide, viewport scale
    - But those can be done in a fixed-function pipe
- Data Types and Hardware Support for them
  - 32 bit Single Precision FP scalars (same format as IEEE, but...)
  - 4 way FP SIMD vector (points, colors, normals)
    - Hardware has no-overhead "swizzling" of vector elements
      - Rotate of vector elements (good for fast cross-product)
      - Replication to convert Scalars into Vectors
      - Can make constants [-1,0,1,2] -> [0,0,0,1] or [-1,-1,-1,0]
      - Could replace 2 or 3 "regular" SIMD instructions
    - Write mask bits on all instruction writes
    - Hardware support for no-overhead negation of value
  - Integer used in address register (constant index)
  - Some kind of conversion hardware from byte/short/int to float



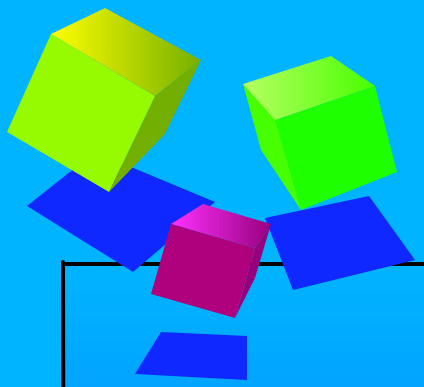
Programming Model (cont)

- 5 Register sets (input, output, constant, GPR, address)
- Vertex Program Input register bank
  - A.K.A "attribute register" bank, "vertex attribute register" bank
  - Composed of 16 quadwords, one read port (that the Vertex Shader sees)
  - Registers have defined functions
    - In fixed-function mode, they seem strictly defined
    - In program mode, seems more like an ABI
  - Registers initialized to (0.0, 0.0, 0.0, 1.0) - saves bandwidth
- Program output (pixel shader input)
  - Strictly defined registers (next stage is fixed-function pipe)
    - Some registers (colors) automatically clamped to (0.0, 1.0)
    - Initialized to (0, 0, 0, 1) at start of program
- System has a very dataflow-ish / streaming nature
  - Program starts on write of attribute register 0
  - Stops when output register 0 written (does this start the pixel shader?)



Attribute Registers

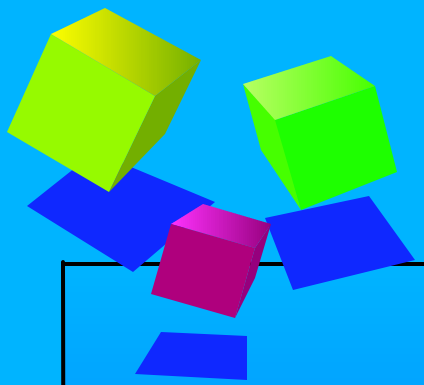
Vertex Attribute Register	Conventional Per-Vertex Parameter	Conventional Per-Vertex Parameter Command	Conventional Component Mapping
0	Vertex Position	glVertex	*x, y, z, w*
1	Vertex weights	glVertexWeightEXT	*w, 0, 0, 1*
2	Normal	glNormal	
3	Primary Color	glColor	*r, g, b, a*
4	Secondary Color	glSecondaryColorEXT	*r, g, b, a*
5	Fog Coordinate	glFogCoordEXT	*f, 0, 0, 1*
6	-	-	-
7	-	-	-
8	Texture Coordinate 0	glMultiTexCoordARB (GL_TEXTURE0...)	*s, t, r, q*
9	Texture Coordinate 1	glMultiTexCoordARB (GL_TEXTURE1...)	*s, t, r, q*
10	Texture Coordinate 2	glMultiTexCoordARB (GL_TEXTURE2...)	*s, t, r, q*
11	Texture Coordinate 3	glMultiTexCoordARB (GL_TEXTURE3...)	*s, t, r, q*
12	Texture Coordinate 4	glMultiTexCoordARB (GL_TEXTURE4...)	*s, t, r, q*
13	Texture Coordinate 5	glMultiTexCoordARB (GL_TEXTURE5...)	*s, t, r, q*
14	Texture Coordinate 6	glMultiTexCoordARB (GL_TEXTURE6...)	*s, t, r, q*
15	Texture Coordinate 7	glMultiTexCoordARB (GL_TEXTURE7...)	*s, t, r, q*



Output Registers

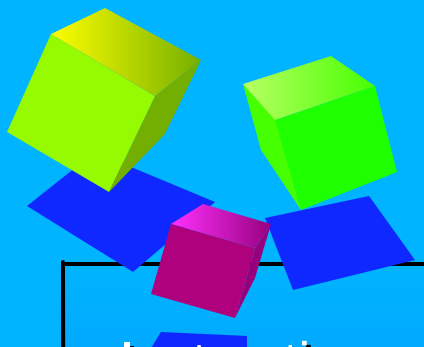
Mnemonic	Full Name	Description
HPOS	Homogenous Clip Space Position	*xyzw*
COL0	Diffuse Color	*rgba*
COL1	Specular Color	*rgba*
FOGP	Fog Distance	*f***
PSIZ	Point Size	*p***
TEX0	Texture coordinate 0	*strq*
...	...	...
TEX7	Texture coordinate 7	*strq*

How are vertices are reassembled into polygons?



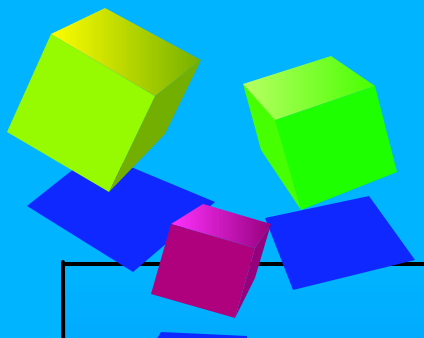
Programming Model (cont)

- Constant bank of 96 quadwords
  - Loaded before the vertex program - not writable by program
  - User definable - matrices, lights, plane coefficients, etc.
  - Made large enough for "indexed skinning" (?)
  - Also single ported
- One address register
  - For indexing into the constant register map
  - Out-of-bounds values return 0s
- Internal Register set of 12 quadwords
  - The "General Purpose Register" set
  - 3 read ports, 1 write port
  - Initialized to 0s when program begins



Programming Model (cont)

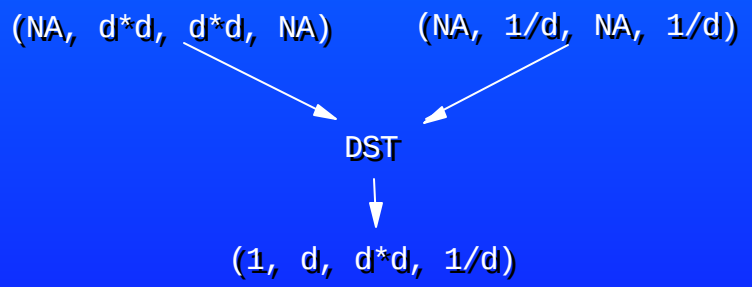
- Instruction set of 17 instructions
  - Initially derived from profiling of fixed-function pipeline
    - 50% of instructions were MUL, ADD, MAD
    - 40% of instructions were dot products
  - All instructions required to have the same latency
    - Pipe control becomes trivial
      - Multithreading control could be almost free
      - No instruction scheduling, no dependancies, no hazards
    - But puts a limit on individual instruction complexity
    - No breaking the pipeline - divide/square root/denormilization
    - What about LIT and DST? (yes - same latency)
    - Reminicent of RISC/CISC tradeoff
  - All instructions have 4 element write-mask
- Control Flow
  - It's simple - there is none!
  - Programs are limited to 128 consecutive instructions
  - If/else implemented by sum-of-products



Compatibility Issues

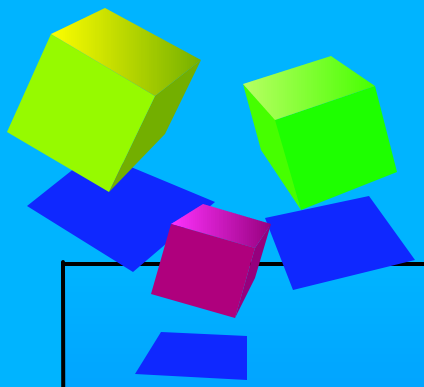
- Previous architecture had lighting engine
  - Heavily lit scenes had 2x speedup in fixed-function mode
  - Add complex instructions (faster than a RISCy vertex program)
    - DST - For constructing attenuating factors
      - Used for: (K0, K1, K2) • (1, d, d*d) = K0 + K1*d + K2*d*d
      - d is some distance
      - 1/d and d*d are natural byproducts of vector normalization
    - LIT - Does ambient, diffuse, and specular lighting
      - Replaces a ~10 instruction vertex program
      - Had to add LOG and EXP hardware (expose these to ISA)

DST



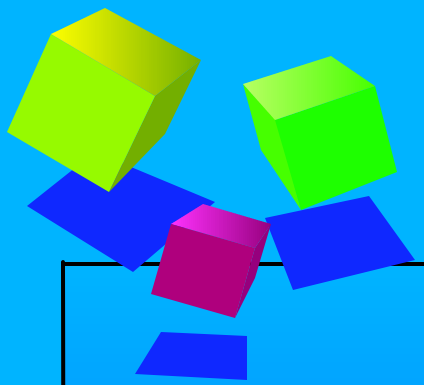
LIT

```
output.x = 1.0 ;                    // ambient
output.y = max(N•L, 0.0) ;          // difuse
output.z = 0.0 ;                    // specular
if ((N•L > 0.0) && (p == 0))
   output.z = 1.0 ;
else if((N•L > 0.0)  && (N•H > 0.0))
   output.z = (N•H)^p ;
output.w = 1.0 ;
```



Instruction Set

Mnemonic	Full Name	Description
MOV	Move	vector -> vector
MUL	Multiply	vector -> vector
ADD	Add	vector -> vector
MAD	Multiply and add	vector -> vector
DST	Distance	vector -> vector
MIN	Minimum	vector -> vector
MAX	Maximum	vector -> vector
SLT	Set on less than	vector -> vector
SGE	Set on greater or equal to	vector -> replicated scalar
RCP	Reciprocal	vector -> replicated scalar
RSQ	Reciprocal square root	vector -> replicated scalar
DP3	3 term dot product	vector -> replicated scalar
DP4	4 term dot product	vector -> replicated scalar
LOG	Log base 2	miscellaneous
EXP	Exp base 2	miscellaneous
LIT	Phong Lighting	miscellaneous
ARL	Address Register Load	miscellaneous



Programming Model - Miscellaneous things

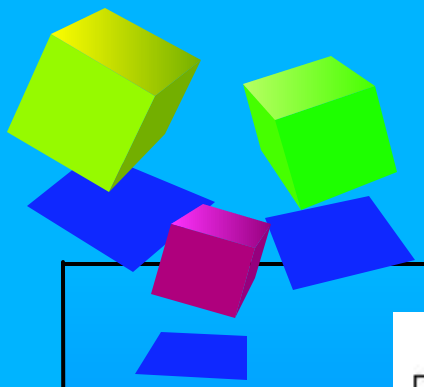
- Not IEEE compatible
  - No denormalized numbers or exceptions
  - Fixed rounding to -infinity
  - 0.0 times X is 0.0, 1.0 times X is X (even if X = NaN or infinity)
  - LOG and EXP are only accurate to 11 mantissa bits
    - Typically operating on 8 bit color values - OK
    - Full precision with ~10 more instructions
  - RCP and RSQ are accurate to about 1.5 bits
- Definitely different than a common CPU
  - The "phong lighting" instruction really gives it away
  - 17 instructions - and none of them are branches
  - Architecture defines ported-ness and inital values of registers
  - Everything is FP - even index into memory! (Get me some water!)

Comments on low-level Programming

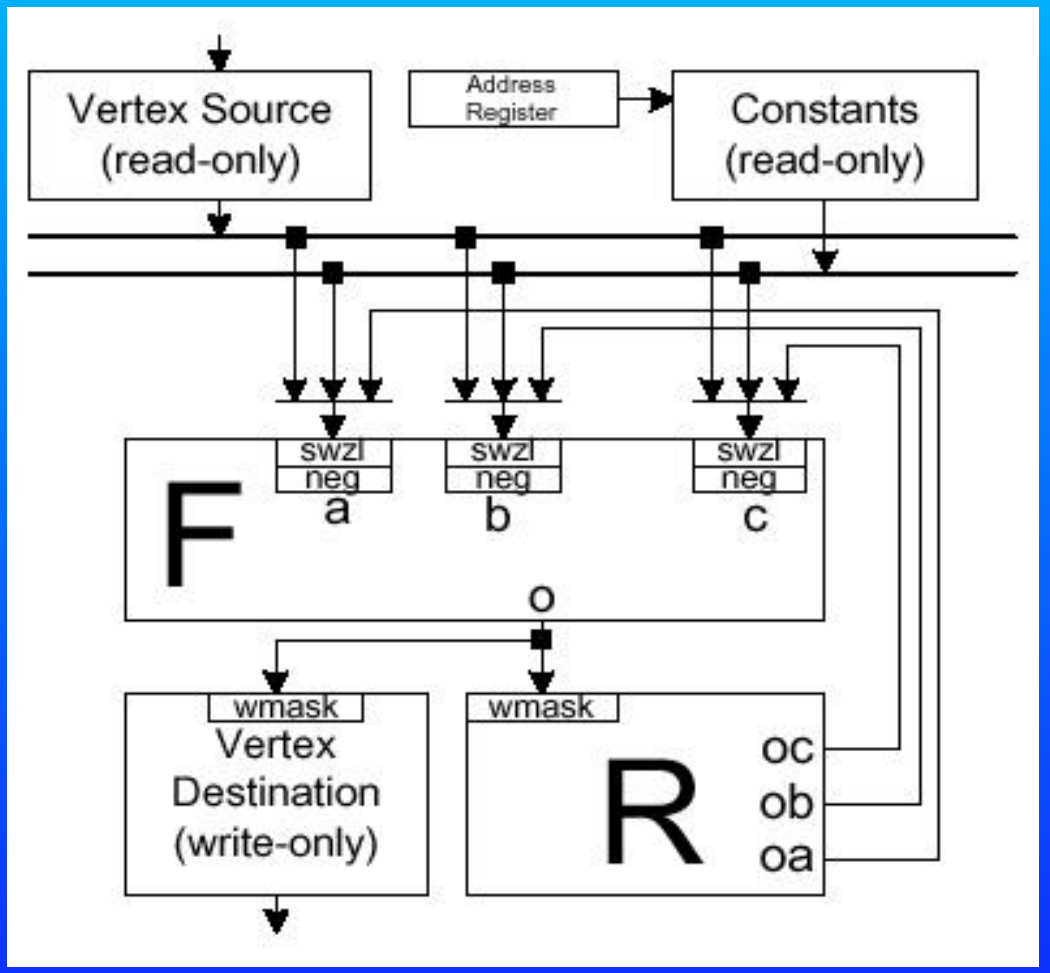
- Programming model/ISA fit the task - Amdahl's Law
  - 1 instruction dot product
  - 2 instruction cross-product (thanks to swizzling and negation)

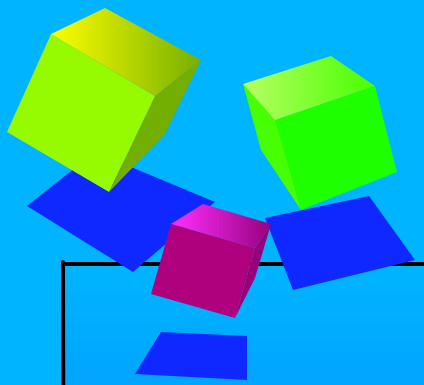
```
MUL R1, R0.zxyw, R2.yzxw ;
MADD R1, R0.yzxw, R2.zxyw, -R1
```

  - Reciprocal is better than divide for many things
    - perspective division / normalization - one RCP and 3 MULs
    - Better than 3 non-pipelined divisions
- Non-orthagonal instruction set
  - Most common operations have similiar sub-operations
  - Can't be slower than the last chip - legacy lighting hardware
  - Instruction set elegance - good for papers, bad for wallet
- Architecture is fairly concrete/exposed
  - Not many abstractions - listed as a design goal
  - There might be some at the system level though



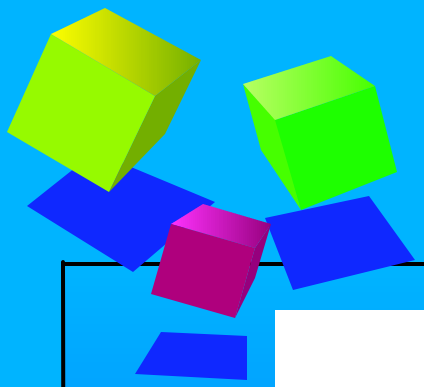
Programming Model - View of the Hardware



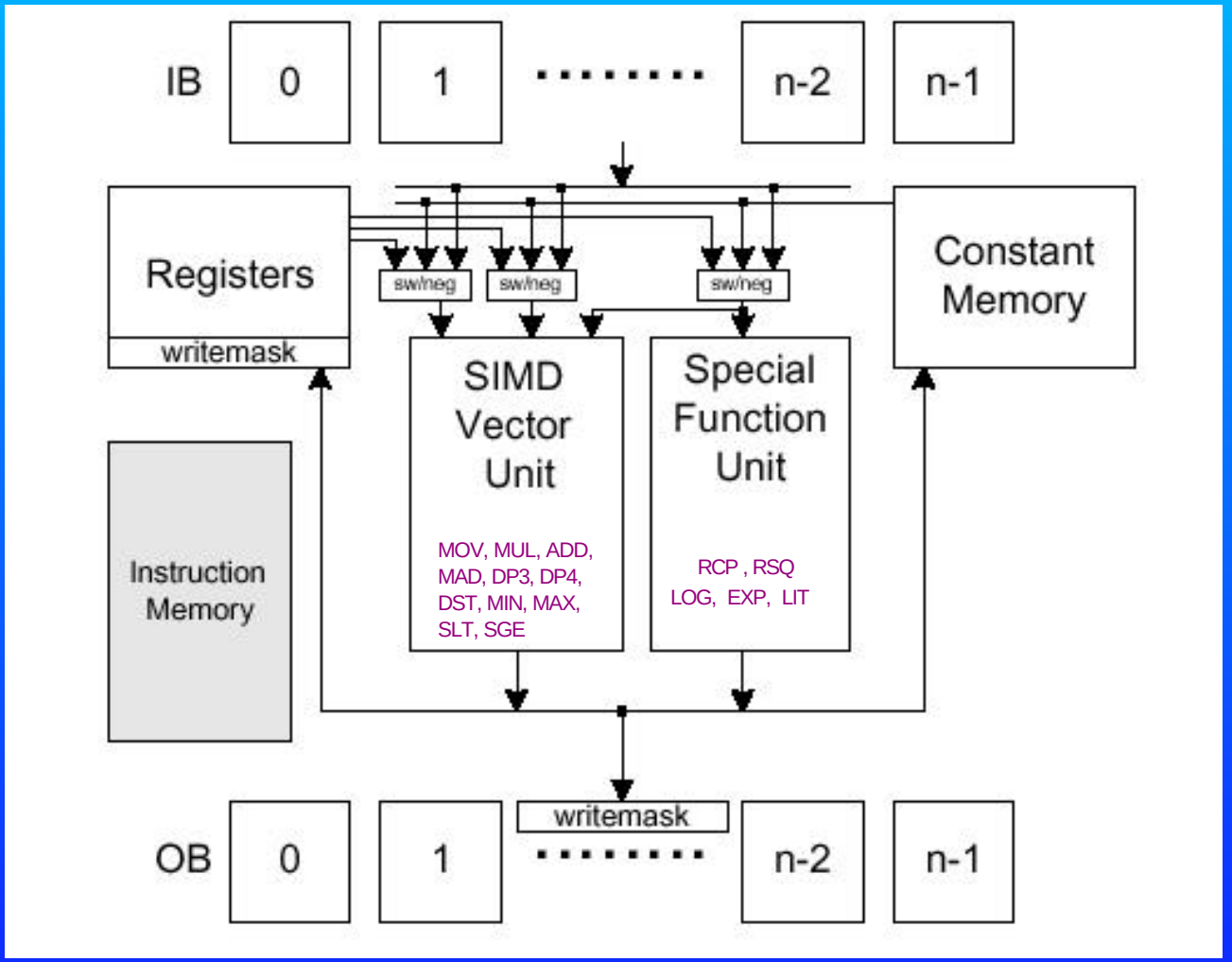


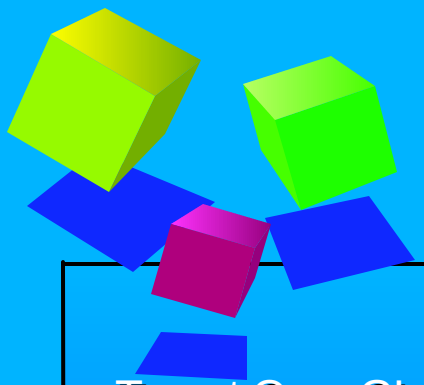
Vertex Shader Hardware

- VAB (Vertex Attribute Buffer)
  - All incoming data converted to floating point before arrival here
  - Input register bank initialized to (0,0,0,1)
  - VAB drains into input buffers
    - One input buffer per thread
    - Input Buffers round-robin into FP processor
- Floating Point Processor
  - Multithreaded Vector Processor (Round Robin)
    - indicates that MT used for hiding pipeline latency
    - Also good at hiding bad programming (no pipe scheduling)
  - Non-IEEE floating point unit
  - 2 pass Newton/Raphson for RCP/RSQ
  - All input registers have fixed and equal timing
- Seems very efficient - very little is devoted to control



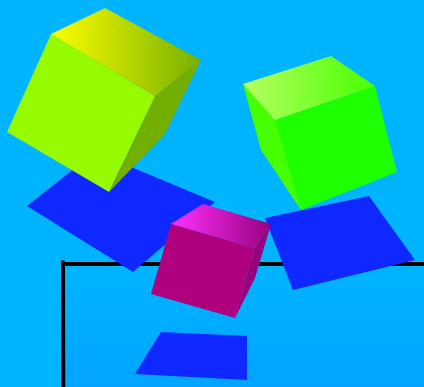
Vertex Shader Hardware





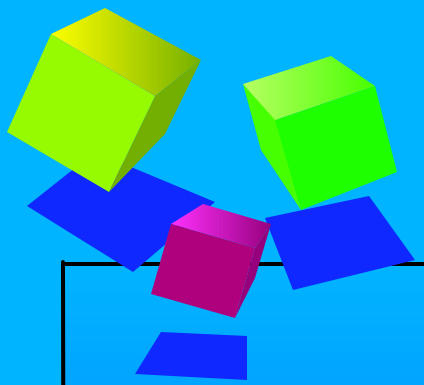
API Design Goals

- Target OpenGL and Direct3D - Widely available, supports easy adoption
  - Extensions need to smoothly blend into current APIs
  - Mix and match new and old functions
  - Don't want to force radical changes onto the programmer]
    - It's nice to give programmer the option to do radical things
    - This is really great - superior things often fail because they're radical
- Need to maintain backwards compatibility
- "Forward Focus" (mentioned in the beginning)
  - Quickly changing API needs programmable hardware
  - Programmable hardware lessens need on fixed API
    - User defines data structures and communication
    - Flexibility to do something really new
    - Automatic generation of vertex programs mentioned (is this done?)
- Path to more programmability (GPU could do more than vertex processing)
- Well defined (constrained) execution environment (This was really a goal?)
  - Not unlimited registers, program lengths, memory
    - This is desirable in General Purpose CPU - hide the implementation
  - Didn't want to overwhelm the programmer with too many degrees of freedom
  - Good from a hardware overhead perspective



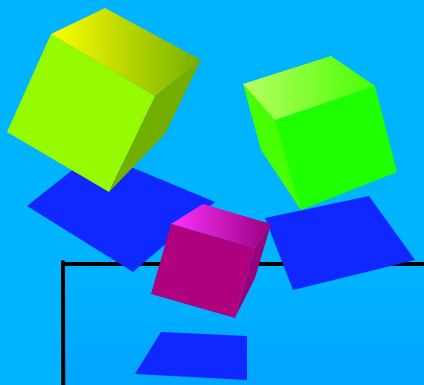
OpenGL programming

- OpenGL - Added a "Vertex Program Mode"
  - Disabled by default, enable with **`glEnable(GL_VERTEX_PROGRAM_NV)`**
  - **`glVertex(...)`** or equivalent command now runs a vertex program
  - Multiple programs managed with "program objects"
    - Treated similiar to texture objects and display lists
      - Program objects have distinct target that indicates its type
      - 2 targets supported
        - **`GL_VERTEX_PROGRAM_NV`** for **`glVertex()`** type programs
        - **`GL_VERTEX_STATE_PROGRAM_NV`** when constants/state needs to be modified
    - Program objects are immutable, but can be reloaded or deleted
    - Generated with **`glGenProgramsNV()`**
    - Deleted with **`glDeleteProgramsNV()`**
    - Loaded with **`glLoadProgramNV()`**
    - Bind with **`glBindProgramNV()`**



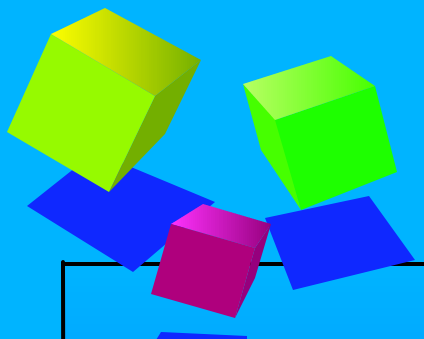
OpenGL programming

- glRequestProgramsResidentNV() loads programs into GPU
  - Performance/caching hint - maybe like the C register type modifier
  - Speeds up binding
- Vertex attributes can change inside or outside **`glBegin()/glEnd()`**
  - **`glVertexAttribNV()`** used to access attribute registers by number
  - **`glVertexAttributesNV()`** specifies a scalar to a usually vector register
  - Can also be accessed by arrays (how isn't discussed)
  - Vertex Attribute Aliasing - assigns names to the attribute registers
    - Minor changes to add shader programs to plain OpenGL code
    - Abstracts for future implementations
- Vertex programs are strings in OpenGL, bytecodes in Direct3D
  - Bytecodes slightly more efficient - almost insignificant
  - Strings more readable - hopefully fewer bugs



OpenGL code

```
static const char programString[] =
"!!VP1.0"
"MOV o[HPOS], v[OPOS] ;"
"END";


// Load program into 7
glLoadProgramNV(GL_VERTEX_PROGRAM_NV, 7,
    strlen(programString), programString);

// Make 7 the current vertex program
glBindProgramNV(GL_VERTEX_PROGRAM_NV, 7);


glBegin();

glVertex(blah);         // Runs the "7" program with blah as input

glEnd();
```



Vertex Programs

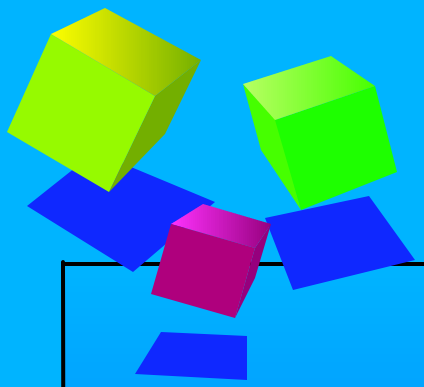
- Program Parameters (96 element memory for constants)
  - There's no automatic aliasing to vertex states (lights, clip planes, etc)
  - But, there is "Matrix Tracking"
    - For both OpenGL and user matrices (modelview, projection, tex, etc)
    - Stored in a dedicated contiguous area of constant memory
    - Can track the transform of a matrix (into constant memory 4,5,6,7):

```
glTrackMatrix(GL_VERTEX_PROGRAM_NV, 4, GL_MODELVIEW, GL_INVERSE_TRANSPOSE_NV)
```

      - Vertex program needs only 3 DP3s to convert to eye space for lighting
    - Can even track a composite of the modelview and projection matrices

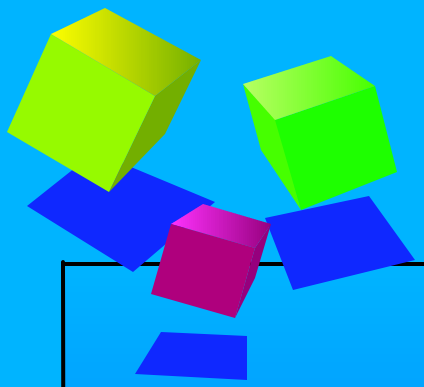
```
glTrackMatrix(GL_VERTEX_PROGRAM_NV, 0,
    GL_MODELVIEW_PROJECTION_NV, GL_IDENTITY_NV)
```

      - Vertex program now needs only 4 DP4s to convert to clip space
  - **`glProgramParameterNV*()`** commands for constant memory setup
  - Parameter memory can't be modified inside of **`glBegin()/glEnd()`**
    - Undefined results for an already running shader program
  - Vertex state program
    - used for updating the parameter registers
    - supported by NV_vertex_program
    - explicitly executed, unlike regular vertex program



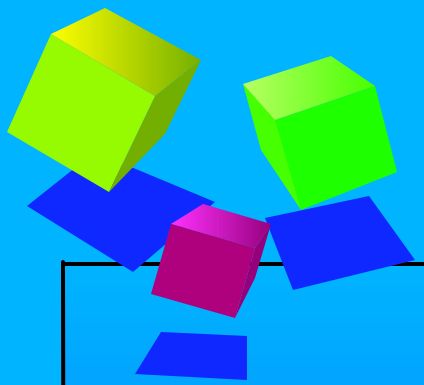
Fin

Thanks!



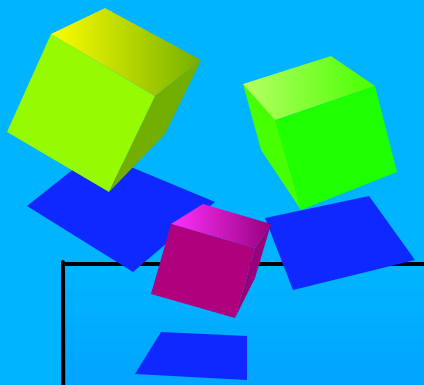
Backup Slides

Backup



Discussion Questions

- Paper 2 finishes by stating their future work will be with increased programmability of geometry processing, programmable fragment processing, and shading languages that exploit GPU programmability - opinions?
- Would adding control flow to Vertex Shader be a good idea? (Radeon has it)
- Lack of hardware details is frustrating - but that's how it goes
- Ideal application domain for multithreading/multiprocessing
  - Enough threads to exeute
  - Effective at hiding pipe latency
  - But - what about memory?
    - Wouldn't be surprised to see more of this in the future
    - Prefetching is probably pretty good in these architectures
    - Some memory accesses are still non-deterministic - MT could help
- GPUs seem to have much lower frequencies than CPUs - why is that?
  - ?? Power/packaging/cost, latch overhead, multithreading, short pipes, complex ops



Guess as to Flap's Memory Tesselation

- Tesselation allows row and column access
- Easy to do with bit manipulation
- Straightforward Address Mapping:

- One Possible Tesselated Mapping: