Master IGMMV
Synthèse d’images et de sons

George Drettakis

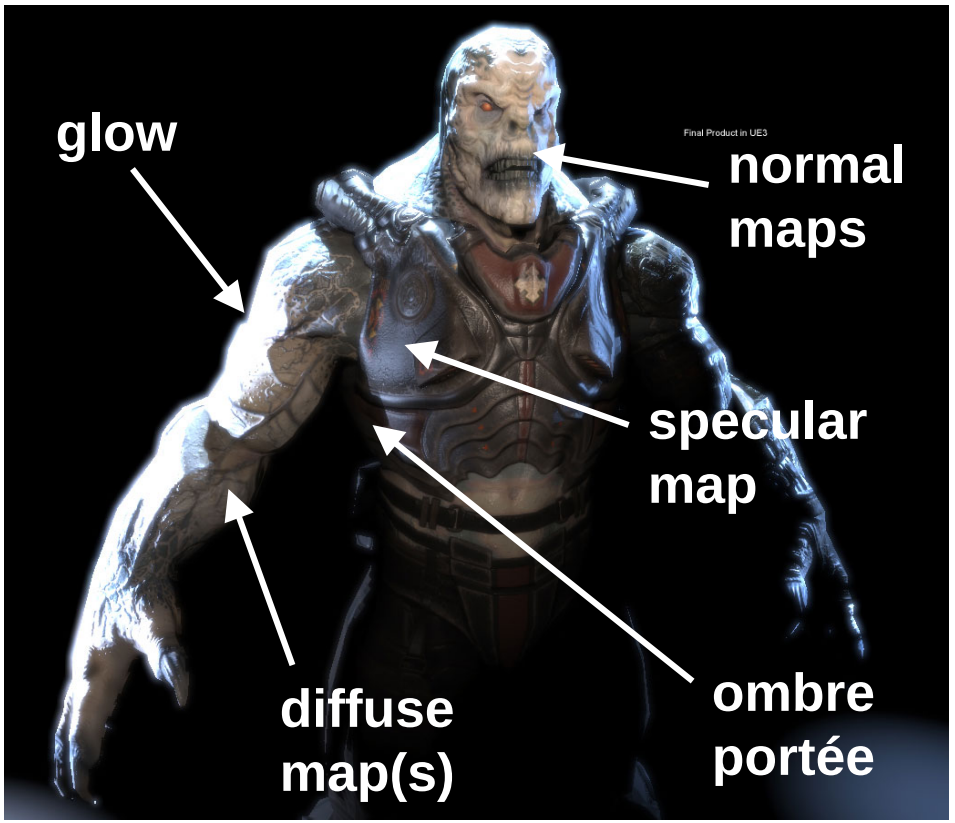
Nicolas Tsingos

Séance 2: Rendu temps-réel avancé

- Modèles d'apparence complexes
  - réflexions, réfraction et « environment maps »
  - shaders
- Complexité géométrique et visuelle
  - niveau de détail et normal maps
  - imposteurs
- Rendu haute dynamique (HDR)
  - capture d'images HDR
  - ré-éclairage avec de la lumière HDR

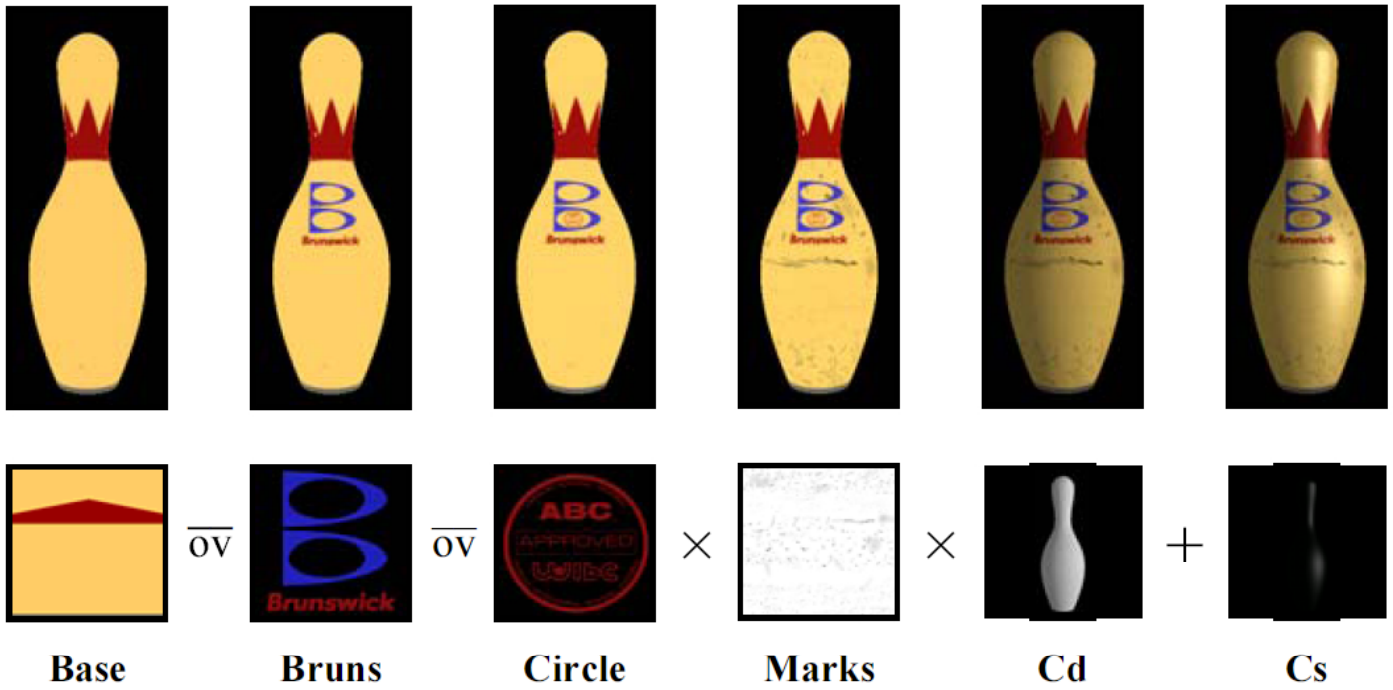
Modèles d’apparence complexes

- Unreal engine



Modèles d’apparence complexes

- “Shaders”



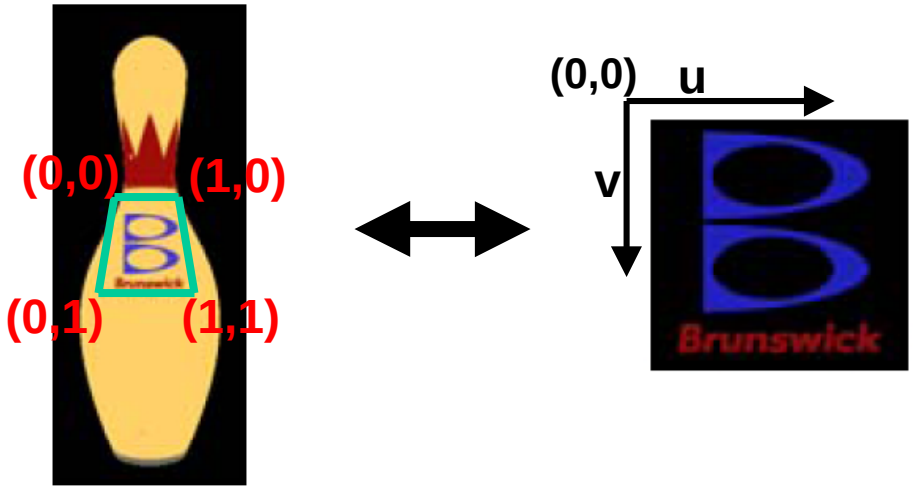
Modèles d’apparence complexes

- Utilisation des textures
  - moduler les couleurs et les matériaux
  - moduler la géométrie
  - moduler et/ou encoder l’éclairage

- Supporté en hardware à travers les vertex/pixel shaders
  - jusqu’à 16 textures par polygone

Moduler les couleurs/matériaux à l'aide de textures

- Utilisation "standard" des textures
  - paramétrisation de la surface (u,v) dans [0,1]
  - la texture donne la couleur diffuse/spéculaire
  - ou également les coefficients de réflexion

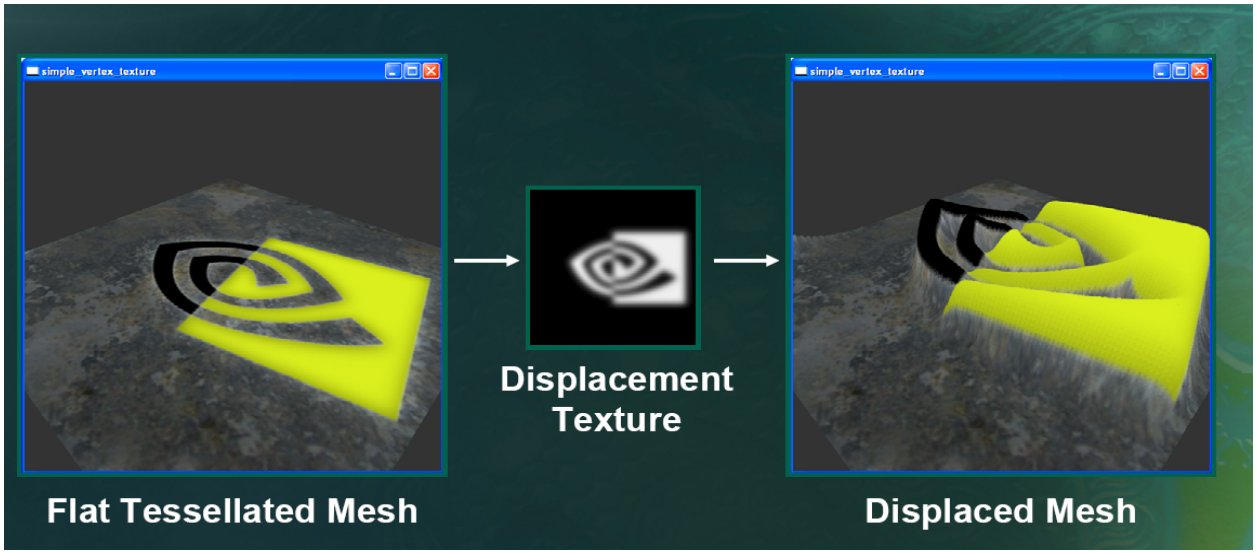


Moduler la géométrie

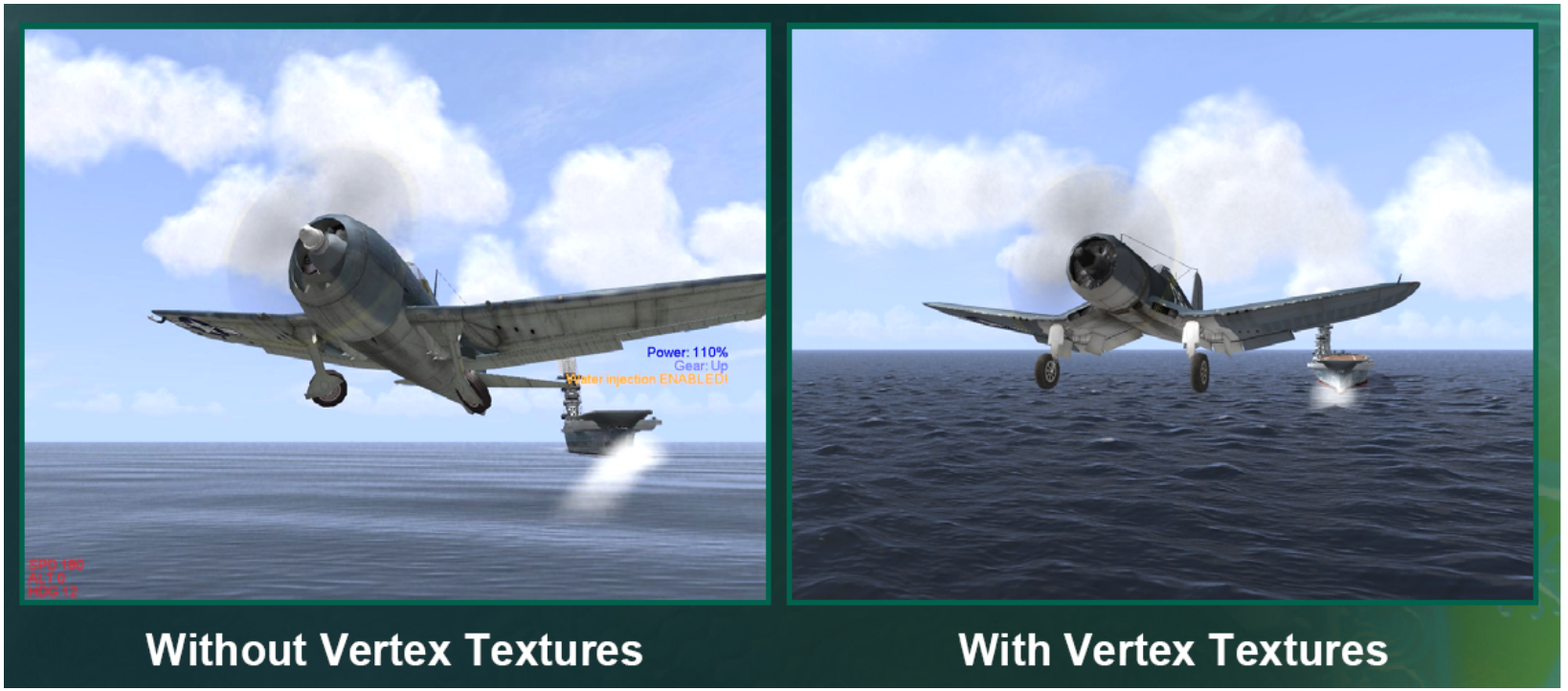
- La texture encode un déplacement 1D autour de la normale ou un déplacement 3D du sommet
- La géométrie est modifiée par un vertex shader
  - “displacement mapping”
  - supporté en hardware par les dernières cartes (jusqu’à 4 textures simultanées)

Displacement mapping

- Avantages :
  - occlusions + silhouettes
  - + facile à modéliser, modèle procédural
- Inconvénients : beaucoup de géométrie



Displacement mapping



Imposteurs

- « Image-based rendering »
  - remplacer la géométrie par des objets simples dont la texture dépend du point de vue.
  - collections de polygones (plans) texturés
  - « billboards »

Imposteurs

« Billboard clouds for extreme model simplification » Xavier Decoret et al. , SIGGRAPH 2003

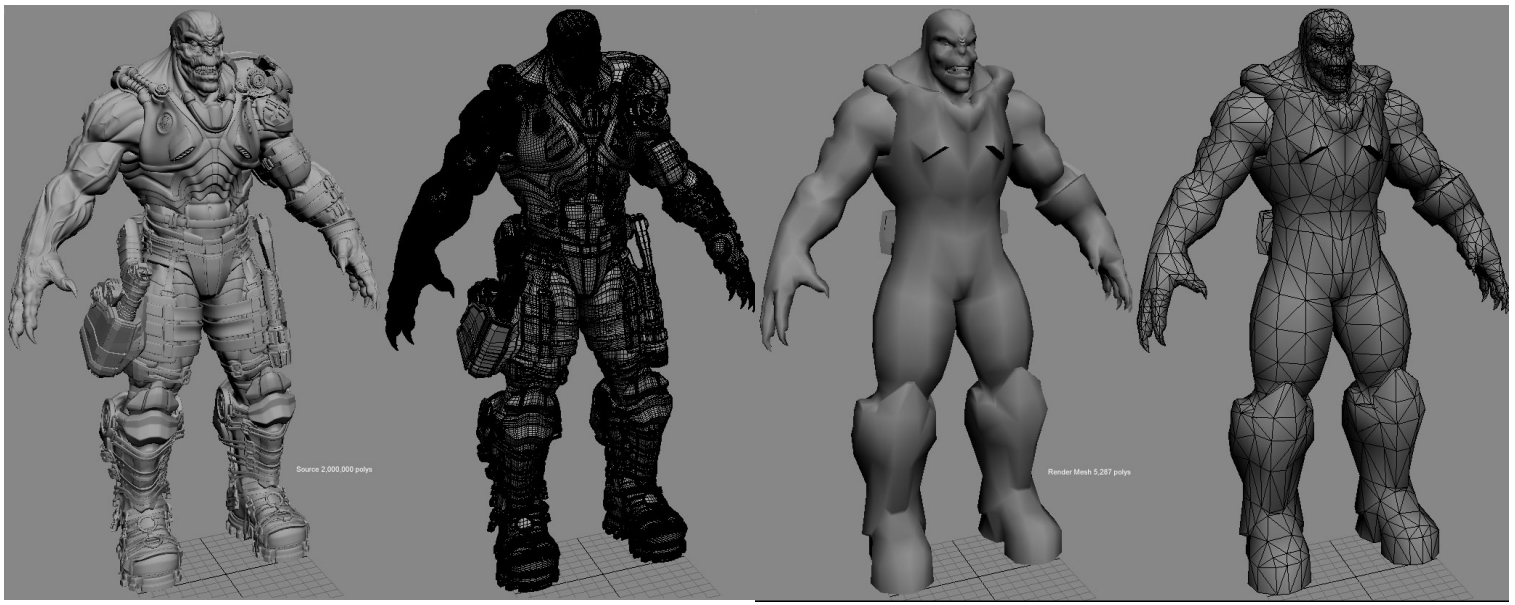
Moduler l’éclairage

- Modification de l’apparence visuelle en conservant une géométrie simple
  - “normal maps”

- Encodage d’un éclairage très complexe
  - non local : réflexions de l’environnement
  - local : du à des sources complexes (e.g., ciel) et aux occlusions (“self-shadowing”)
  - “environment maps”

Complexité visuelle et géométrique

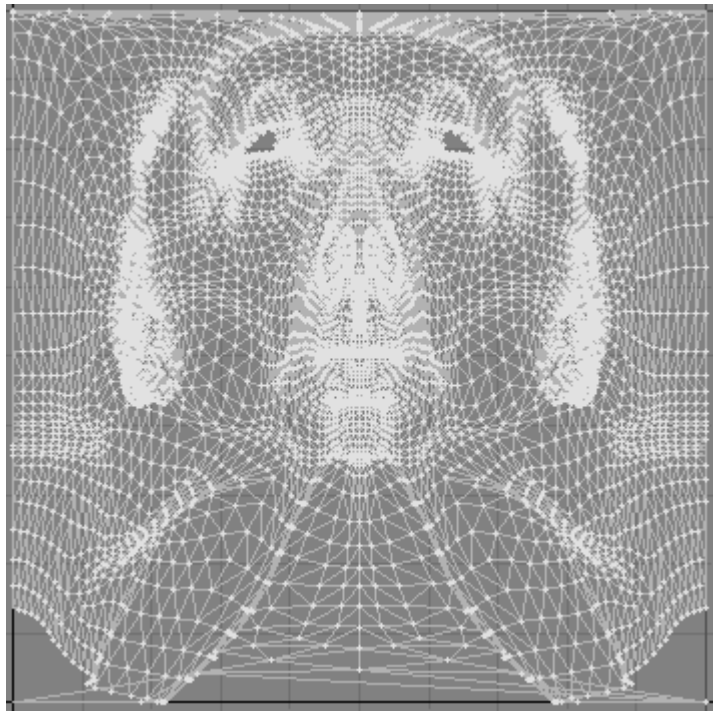
- La complexité visuelle provient d’une combinaison de la géométrie et de l’éclairage

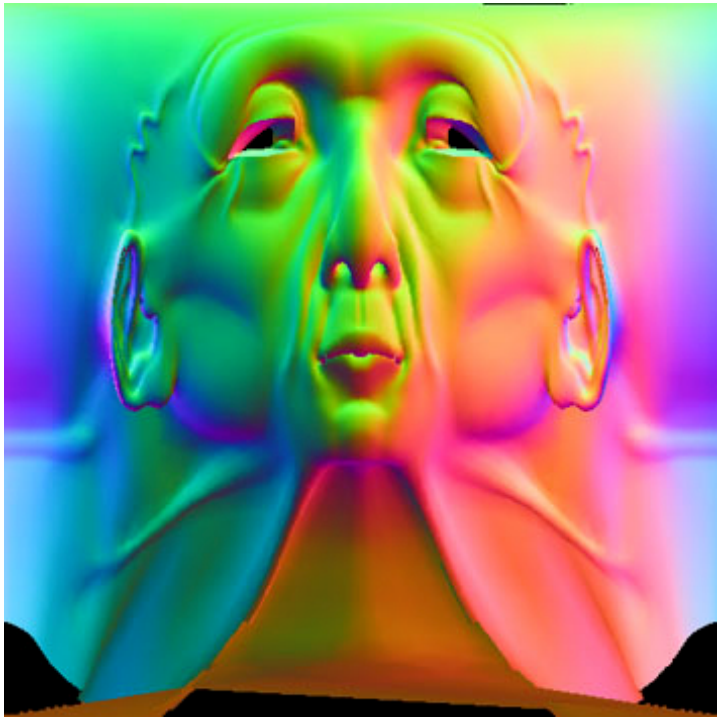


Normal maps

- Encode les normales à haute résolution et la géométrie à basse résolution
  - utilisé quand les effets de parallaxe sont négligeables (petits détails)
  - permet de conserver la complexité de l'éclairage
  - simplifie les silhouettes
  - ne conserve pas les occlusions (auto-ombrage)

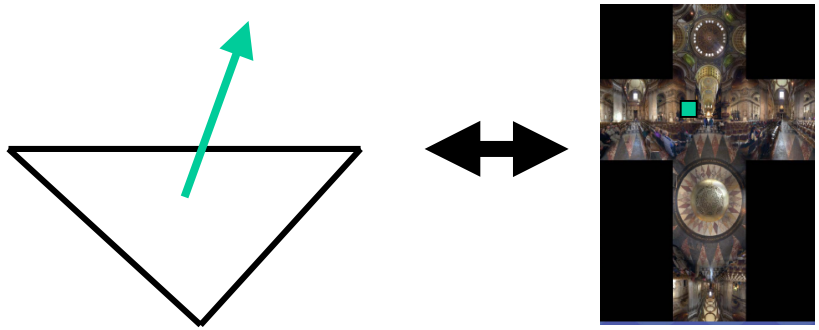
Normal maps





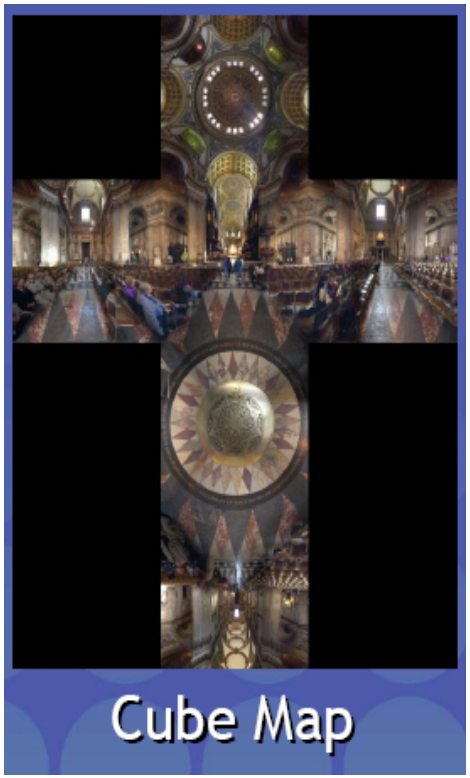
Environment mapping

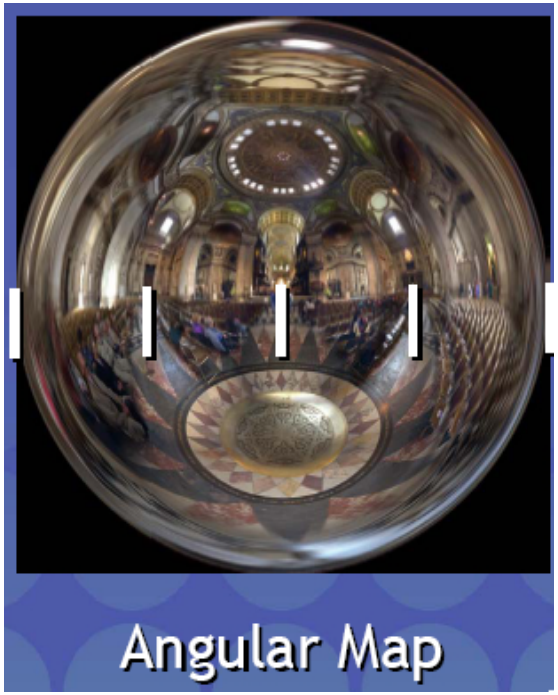
- La texture encode l'environnement autour de l'objet



- Génération de coordonnées de texture "sphériques"
  - glTexGen(GL_S,GL_TEXTURE_GEN_MODE,GL_SPHERE_MAP)
- Accès direct par un vecteur 3D
  - typiquement la direction spéculaire ou la normale

Images omnidirectionnelles





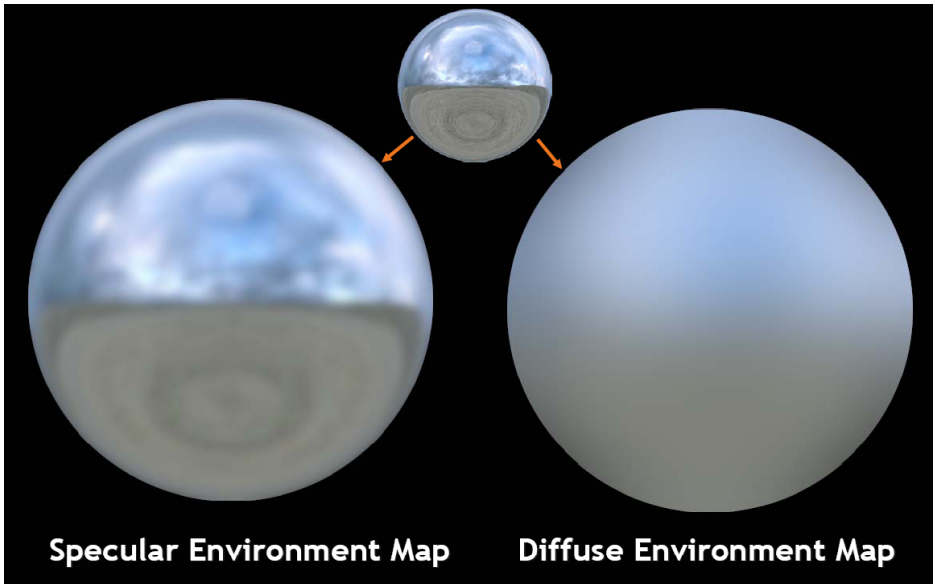
Textures et éclairage local et global complexe

- Importance sampling

Textures et éclairage local et global complexe

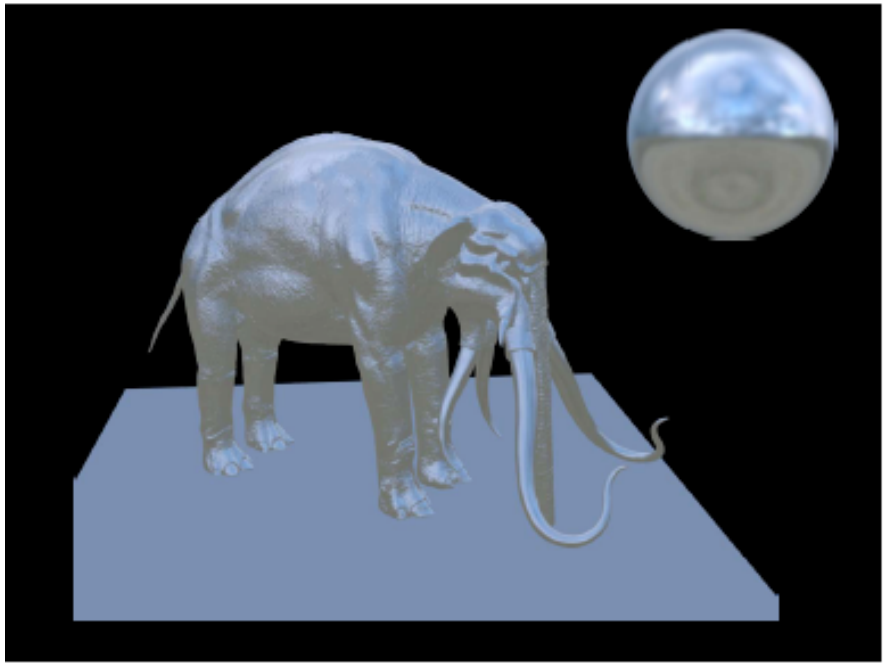
- Sources de lumières complexes
  - ciel
- Maps préconvoluées par la réflectance

$\cos^n(\theta)$ lobe de Phong

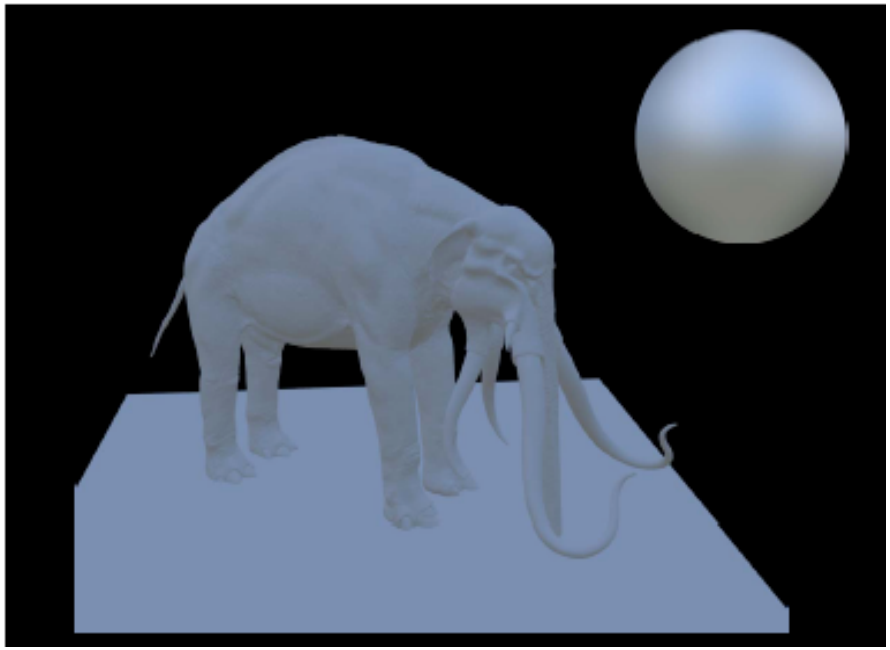


$\cos(\theta)$ diffus

Textures et éclairage local et global complexe

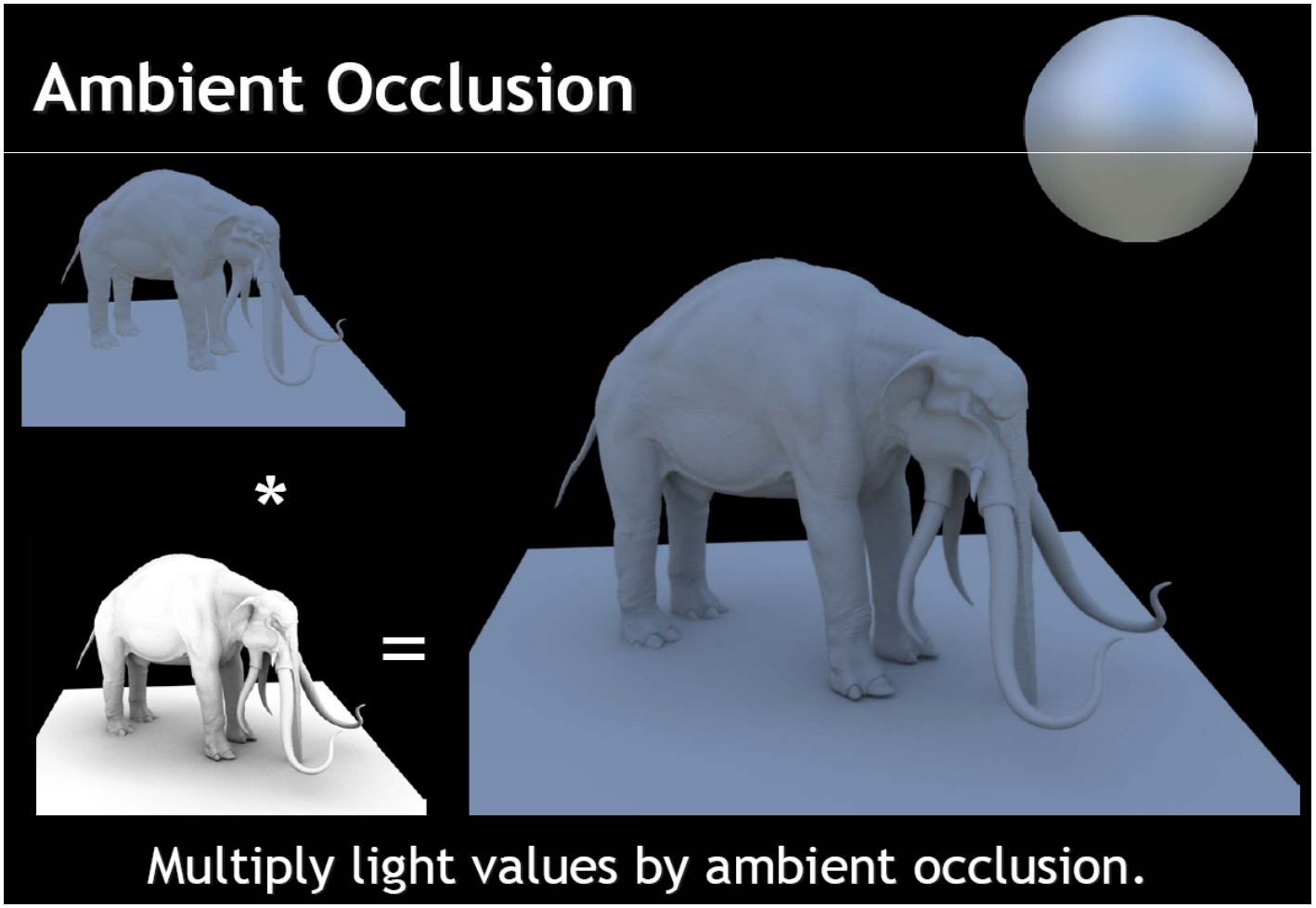


Passe spéculaire :
DIRECTION SPECULAIRE

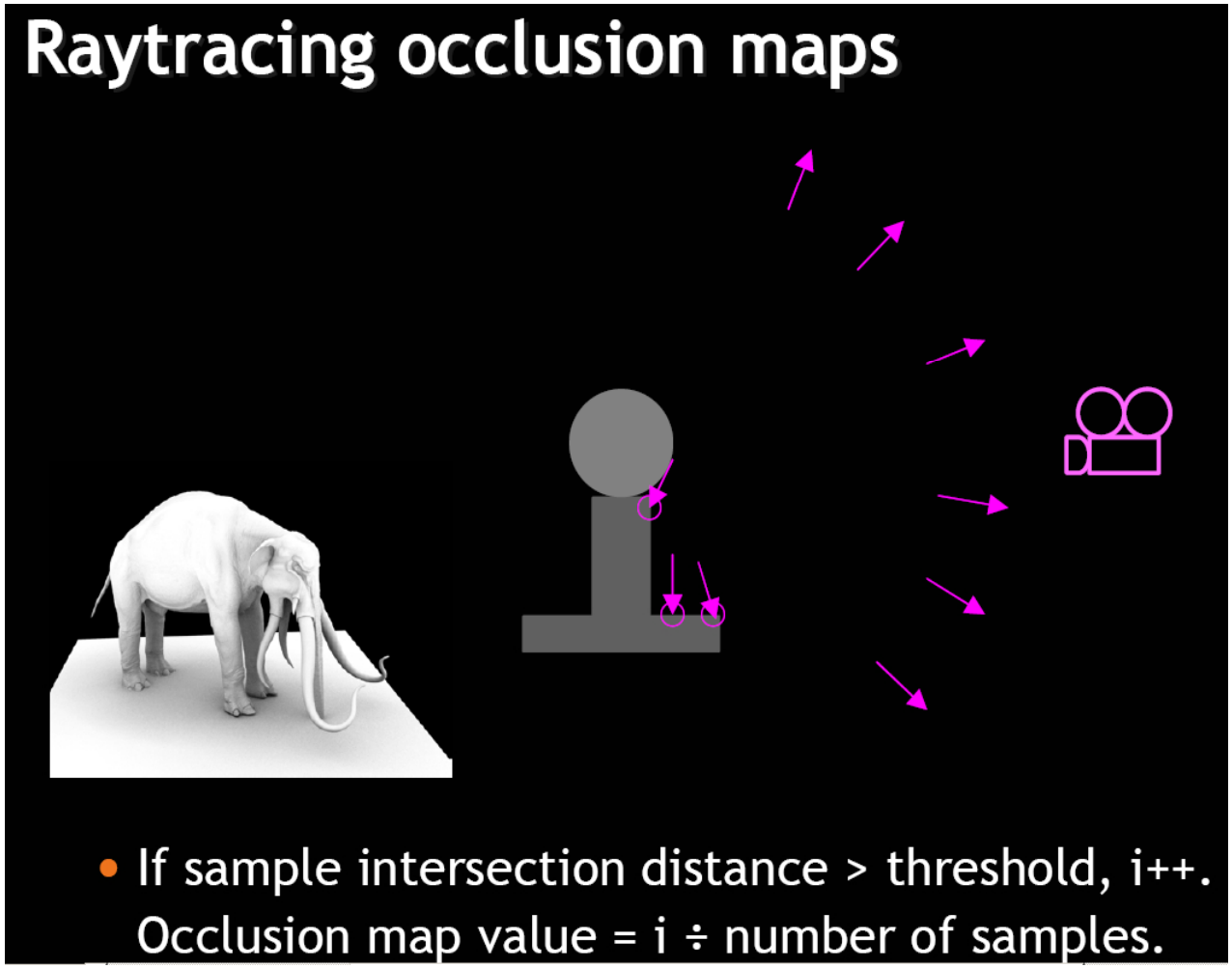


Passe diffuse :
NORMALE

Ombres

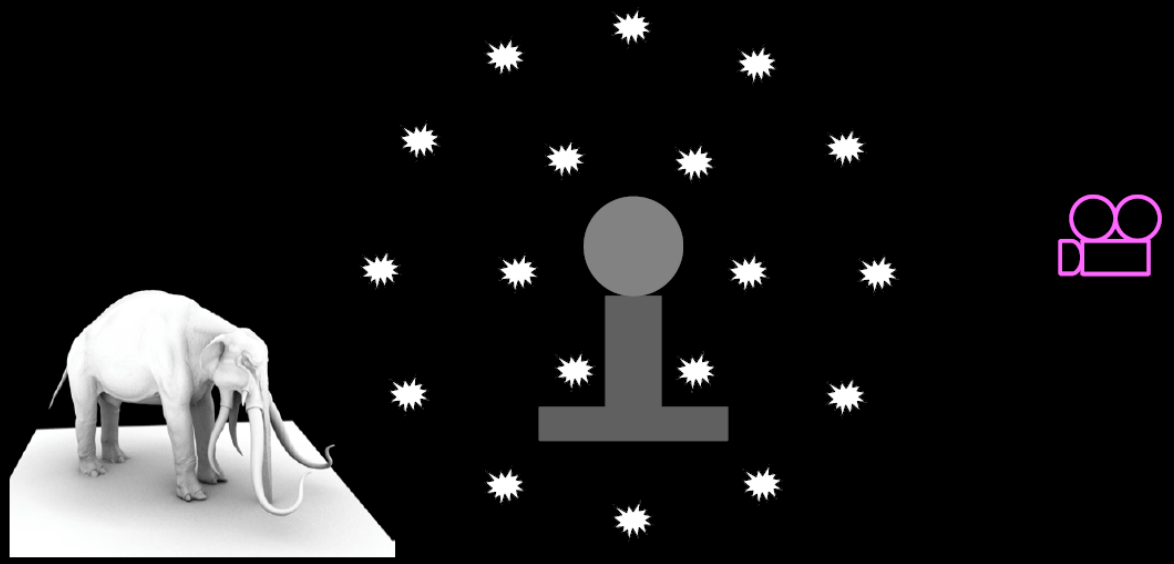

Ambient Occlusion
*
=
Multiply light values by ambient occlusion.

Ombres


Raytracing occlusion maps
• If sample intersection distance > threshold, i++.
Occlusion map value = i ÷ number of samples.

Ombres

Light dome occlusion maps



- Place lights around object in geodesic configuration. Blur z-depth shadows.

Textures et éclairage global

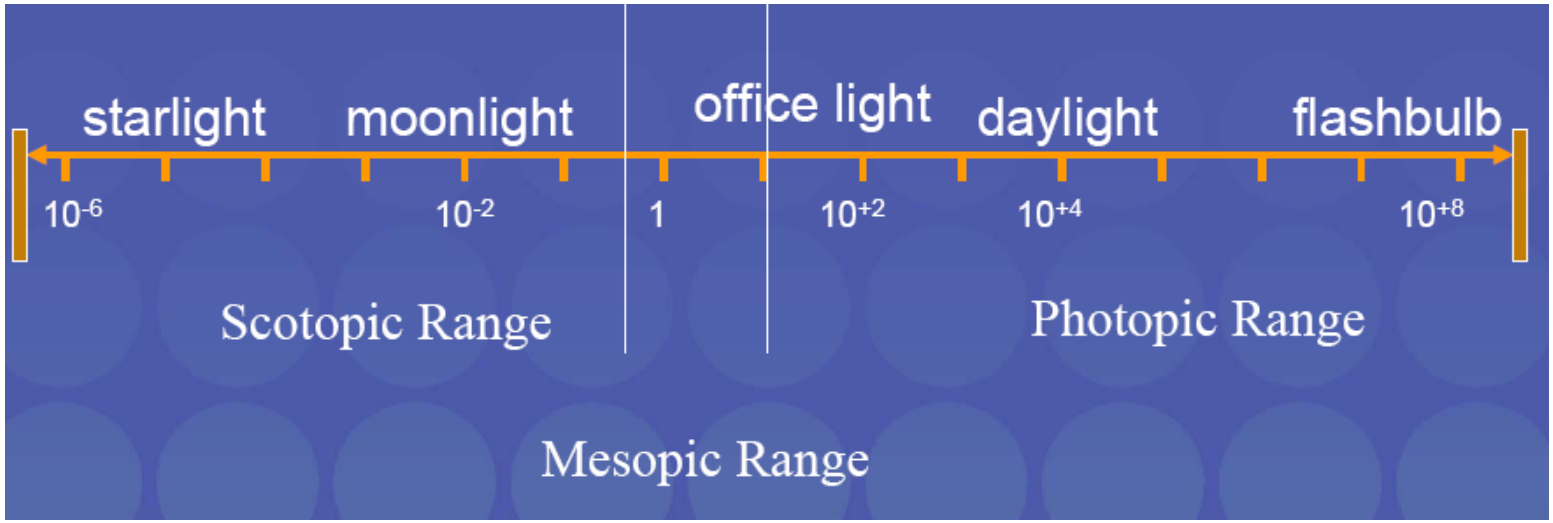
- Précalcul de solutions d'éclairage global indirect (radiosité ou de ray-tracing)
  - stockage en harmoniques sphériques
  - pour des positions de sources variables
  - calcul temps-réel de l'éclairage indirect ou mouvement de la source
- Calcul temps-réel de réflexions ou réfractions
  - échantillonnage de la texture dans la direction réfléchie/réfractée (ou même plusieurs directions)

Liens utiles

- http://www.nvidia.com --> developer
- http://www.ati.com --> developer
- GPU gems (Addison Wesley)

Rendu HDR

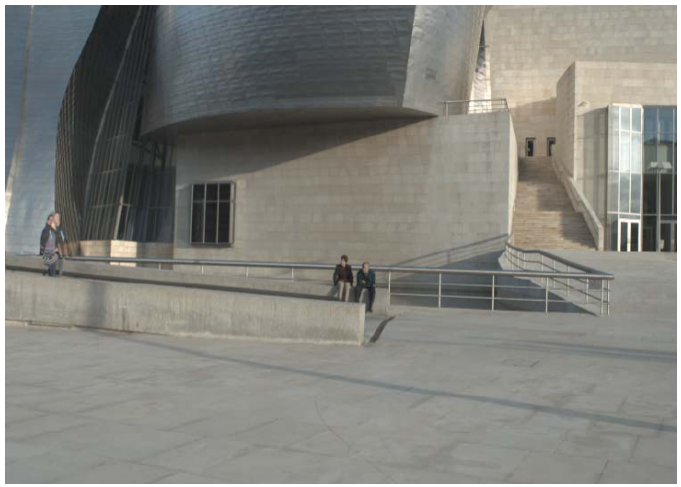
- Pourquoi de l'éclairage haute dynamique ?
  - le monde réel est hdr



Lumen per steradian per meter2 (*lm. m-2.sr-1*)

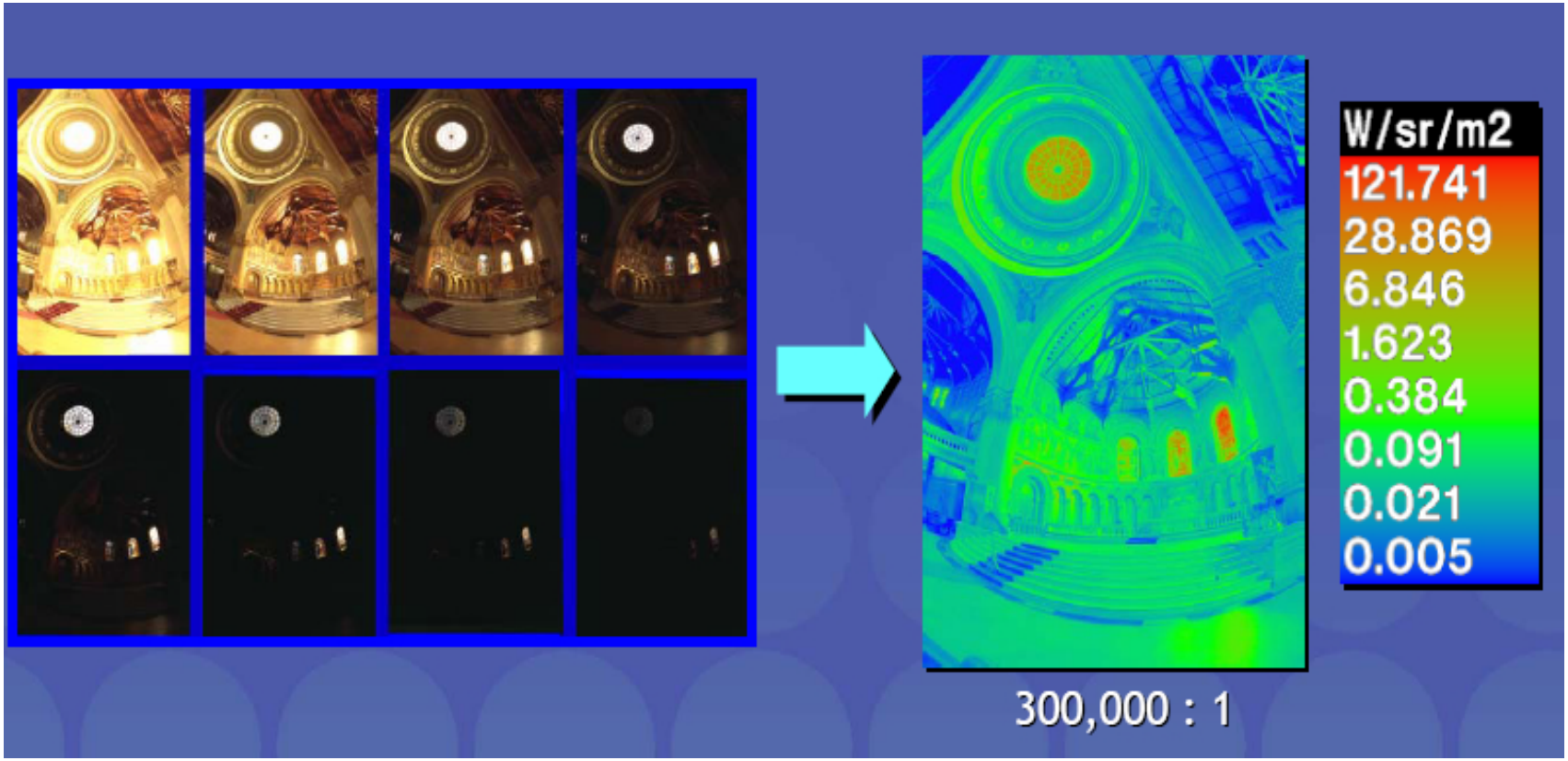
Rendu HDR

- Est-ce si important ?
  - mesurer et représenter finement des sources lumineuses et des propriétés de matériaux
  - on encode plus ces propriétés entre [0,1]
  - parfaite intégration d’objets réels et virtuels !!!



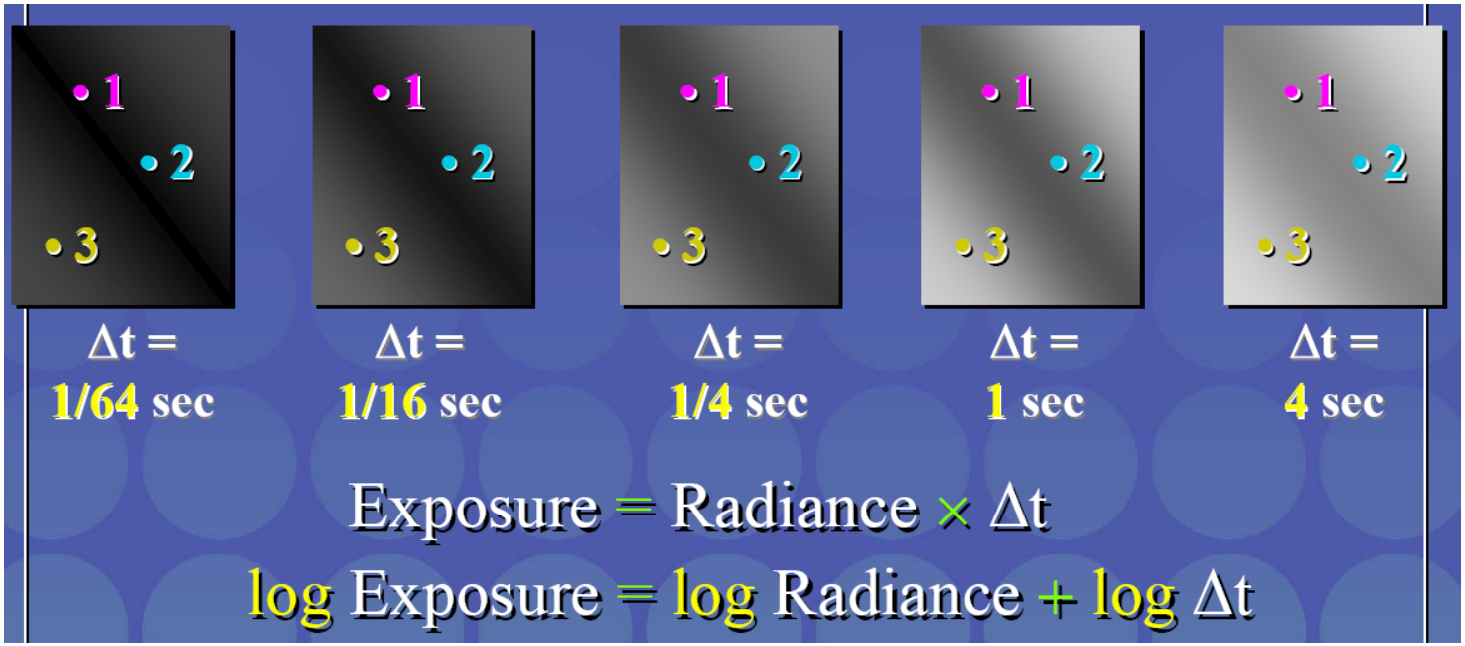
Acquisition d’environnements HDR

- A partir de photos prises à différentes expositions



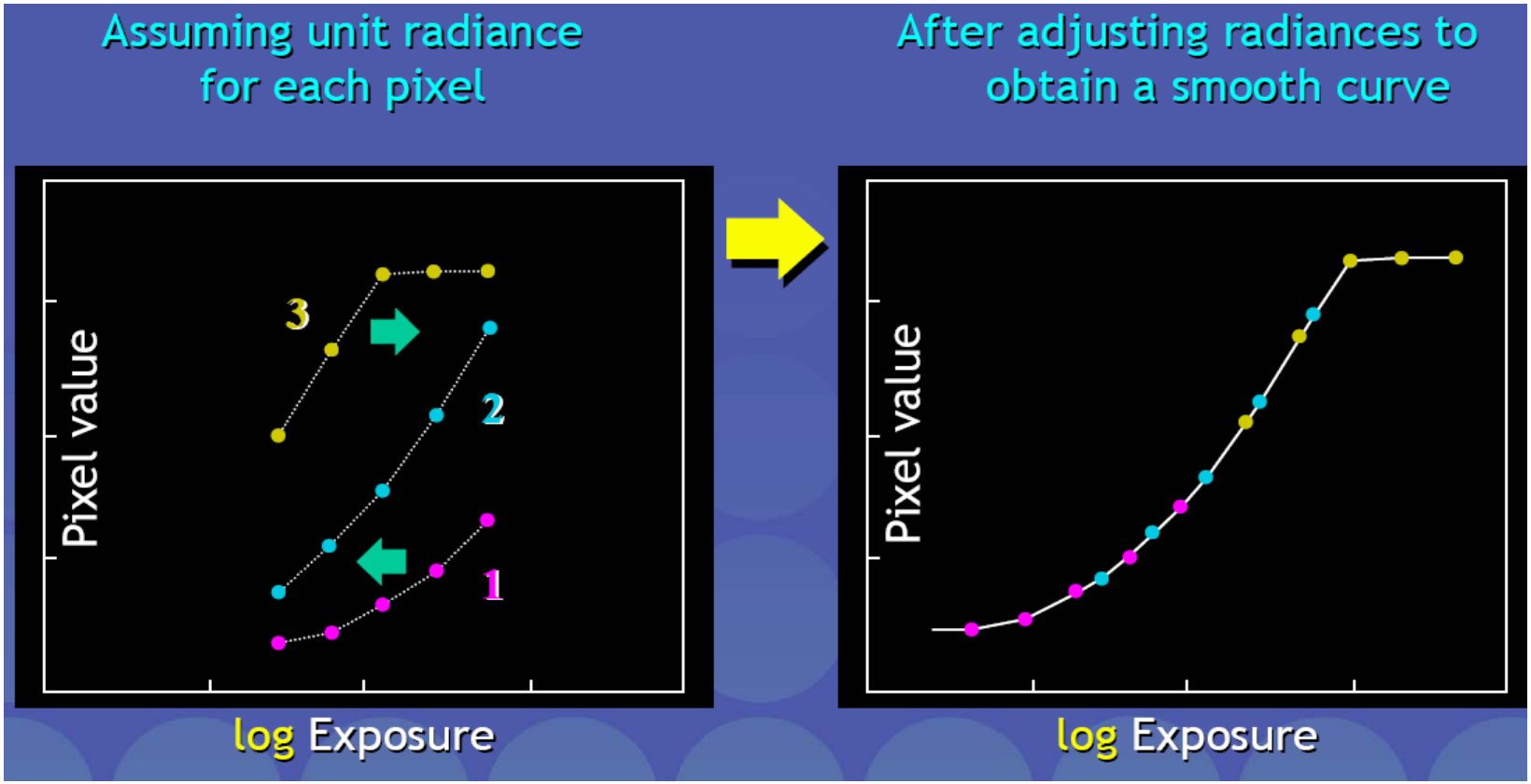
Acquisition d’environnements HDR

- Fonction non linéaire relie la valeur z du pixel à l’exposition : z = f(exposure)

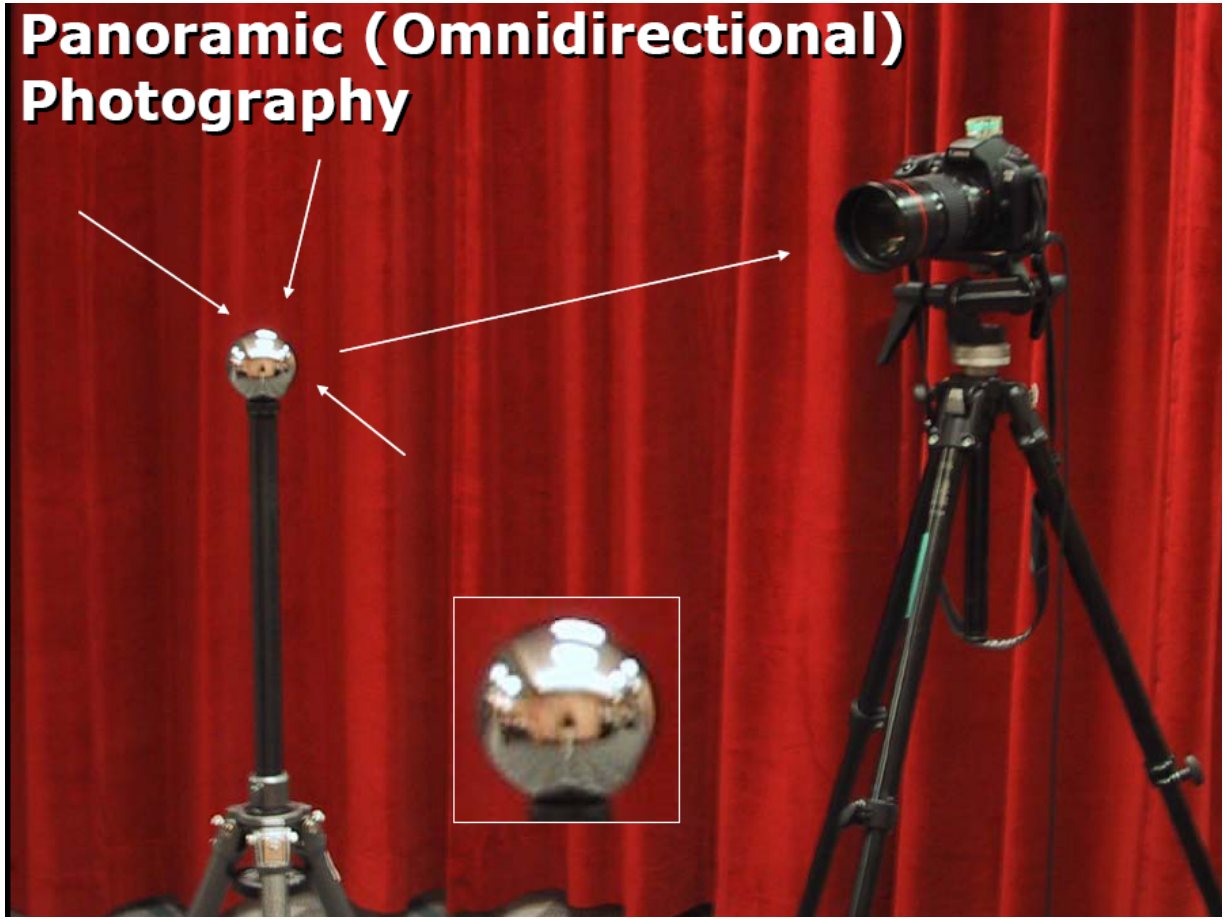


Acquisition d'environnements HDR

- On veut: $g(z) = \ln f^{-1}(z) = \ln E + \ln \Delta T$

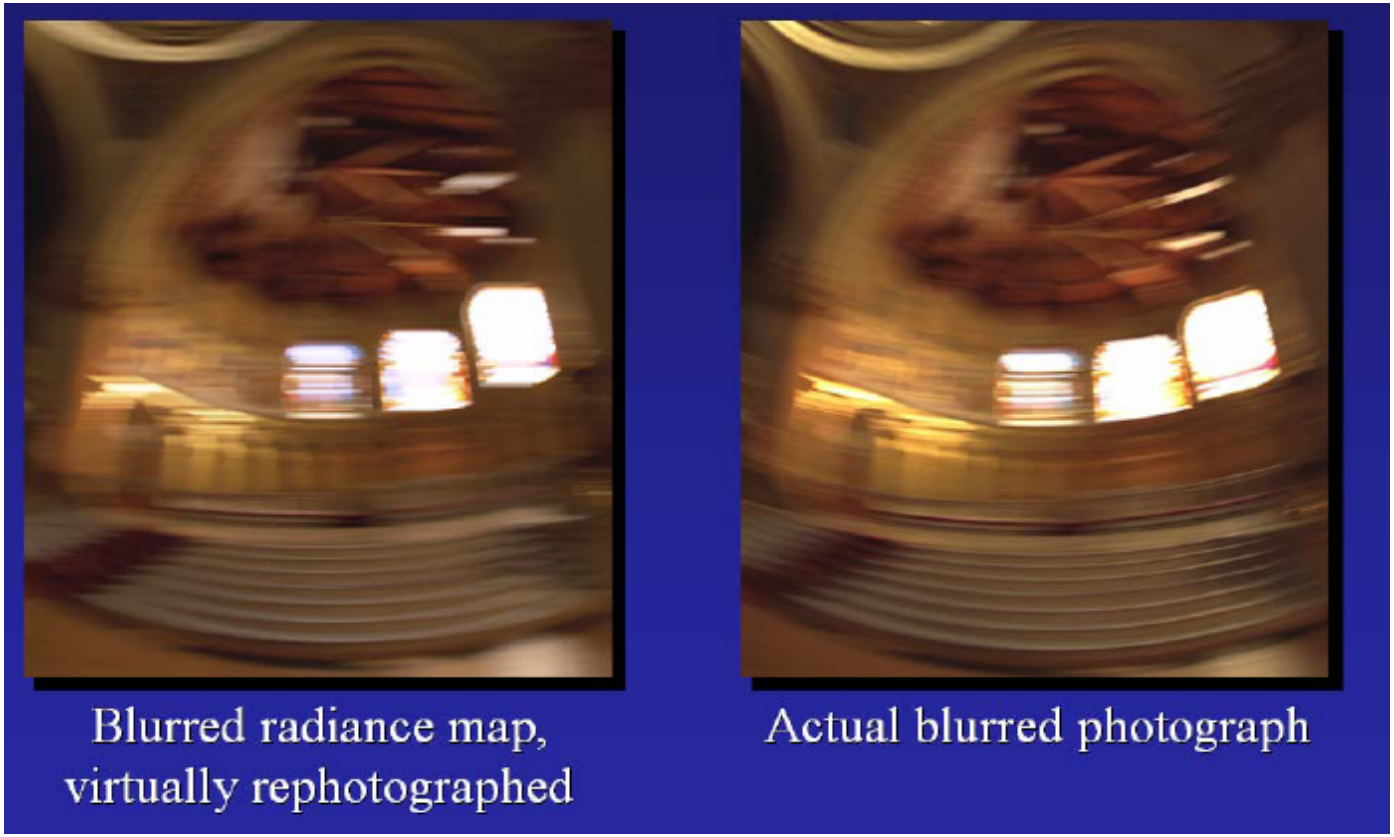


Images HDR omnidirectionnelles



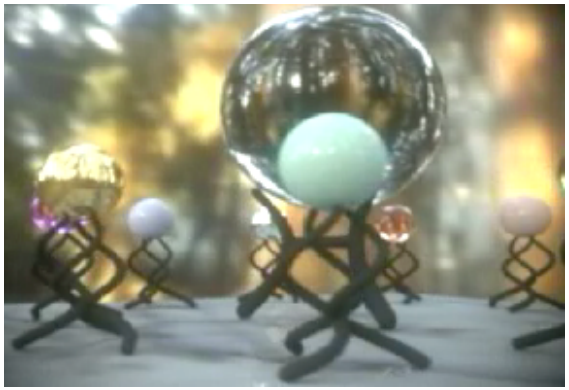
HDR et traitement d’image

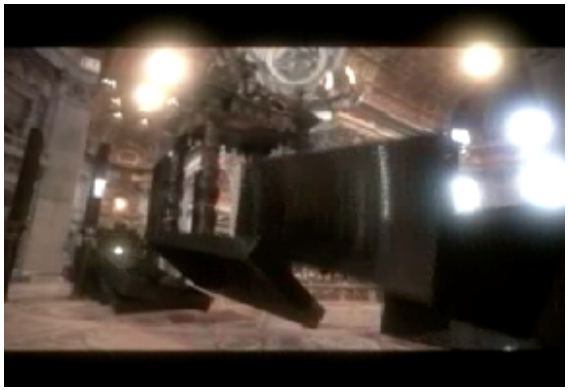
HDR et traitement d’image



Rendu HDR

- Rendering with natural light
- Fiat Lux
- Image-based lighting





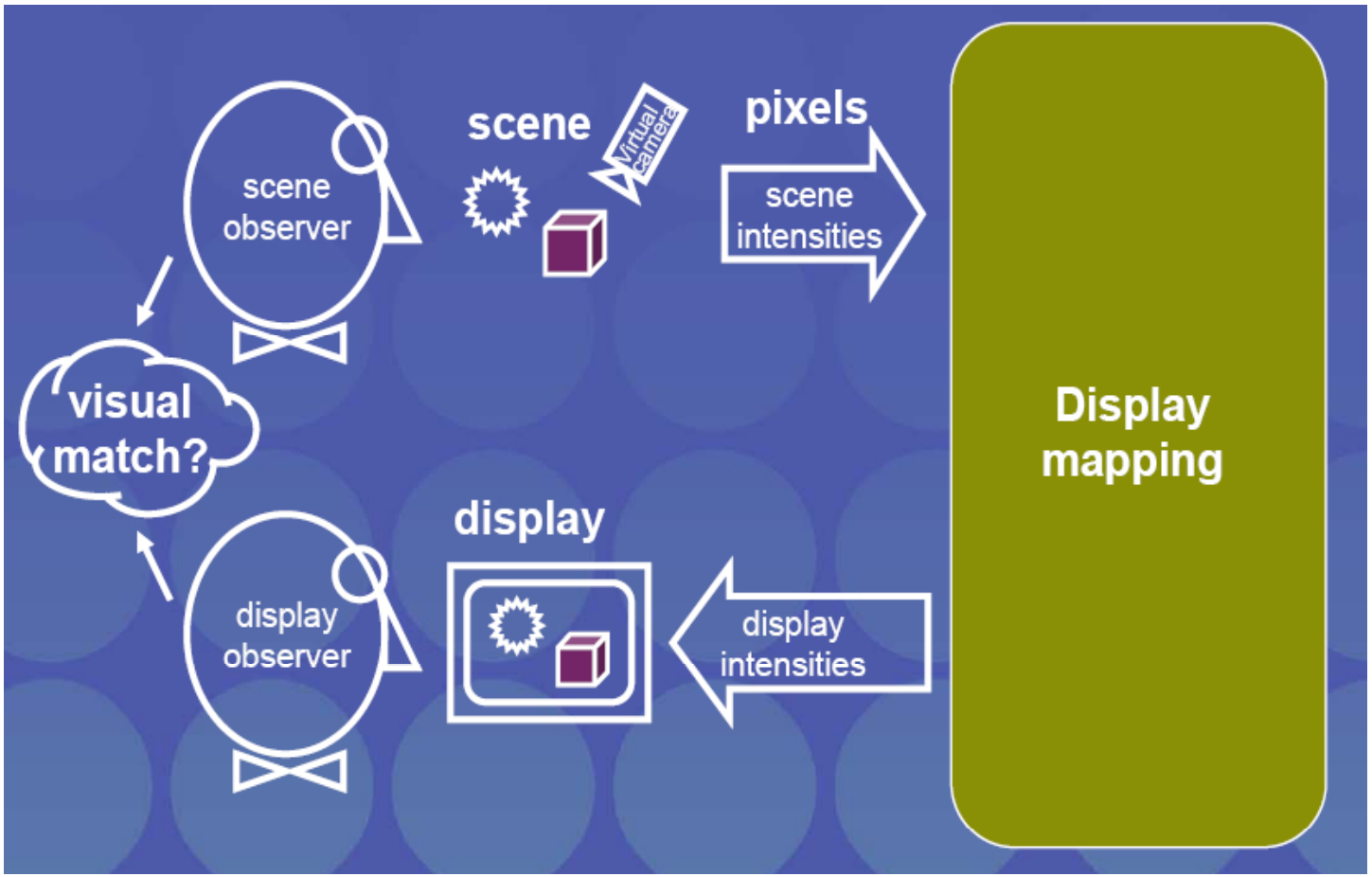
Effets visuels

- Glares, glows, halos
  - produit par les sources de lumière fortes
  - convolution par un filtre
  - appliqué en post-processing sur l'image HDR

Effets visuels

- Mélange d'images
  - "blending"
  - motion blur
- Depth of field

Affichage HDR et “tone mapping”



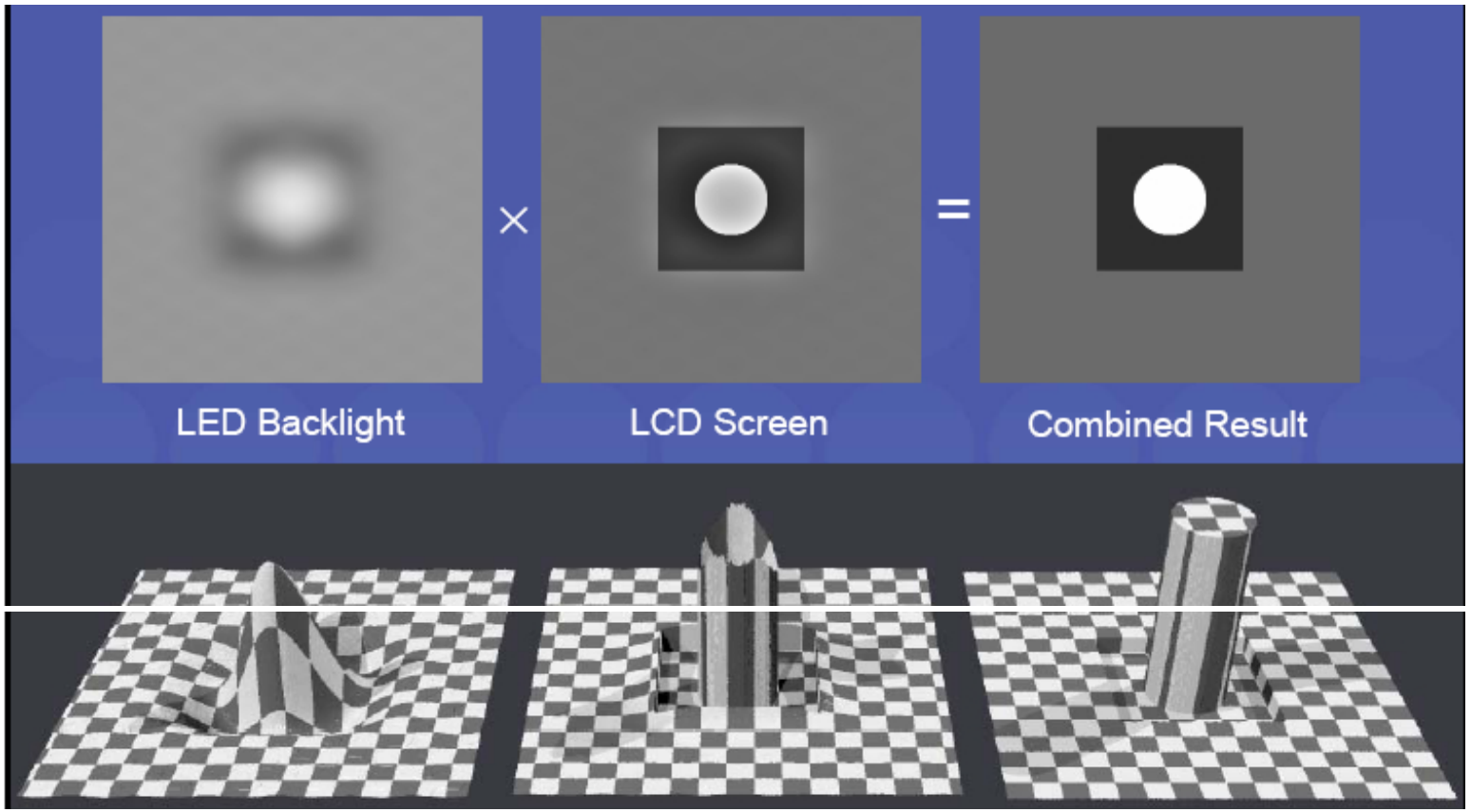
Affichage HDR et “tone mapping”

- Gamma correction
  - new_pixel_value = old_pixel_value^(1.0/gamma)
- Based on image formation
  - Frequency domain
  - Gradient domain
- Based on the human visual system
  - Global operators
  - Local operators

Ecran HDR

- Use Bright Source + Two 8-bit Modulators
  - Transmission multiplies together
  - Over 10,000:1 dynamic range possible

Ecran HDR



Démo

- Masaki Kawase
- Rendu temps-réel
  - textures entières 16 bits ou float 32 bits

Liens utiles

- http://www.debevec.org/Research/HDR
- http://www.debevec.org/HDRShop
- http://www.debevec.org/HDRI2004/
- http://www.daionet.gr.jp/~masa/
- http://www.openexr.org/

Séance 2: Rendu Temps-réel Avancé et Programmation

- Programmation de « shaders »
  - Vertex/Fragment programs
  - CG : C for Graphics (nVidia)
  - cgGL API

Why programmable graphics ?

- Huge consumer demand for compelling visuals
- Per vertex/pixel effects
  - complex custom shading/texturing models
  - image processing / compositing tricks
  - animation
- Off-load CPU
- Limit bus transfers

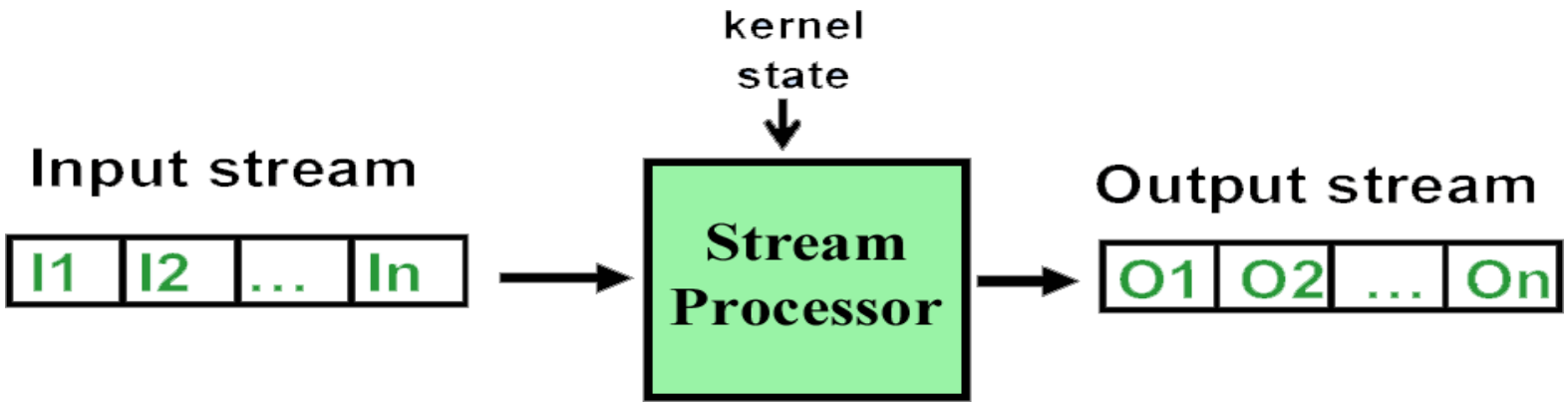
Overview

- Evolution of GPUs
- Low level programming (ASM)
  - OpenGL ARB vertex/fragment programs
- High level Programming (CG/HLSL/GLSLANG)
  - CGgl runtime
  - Applications:
    - per pixel lighting
    - floating point calculations & render-to-texture
- Future

Evolution of GPUs

- Fixed function, mono-texture pipelines
- Fixed function, multi-texture pipelines
- Programmable pipeline
- Earlier programmable graphics HW
  - Ikonas [England 86]
  - Pixar FLAP [Levinthal 87]
  - UNC PixelFlow [Olano 98]
  - Multipass SIMD: SGI ISL [Peercy00]
- RenderMan [Hanrahan90]
- Stanford RTSL [Proudfoot01]

GPU is a « stream processor »



- **The programmable unit executes a computational kernel for each input element**
- **Streams consist of ordered elements**

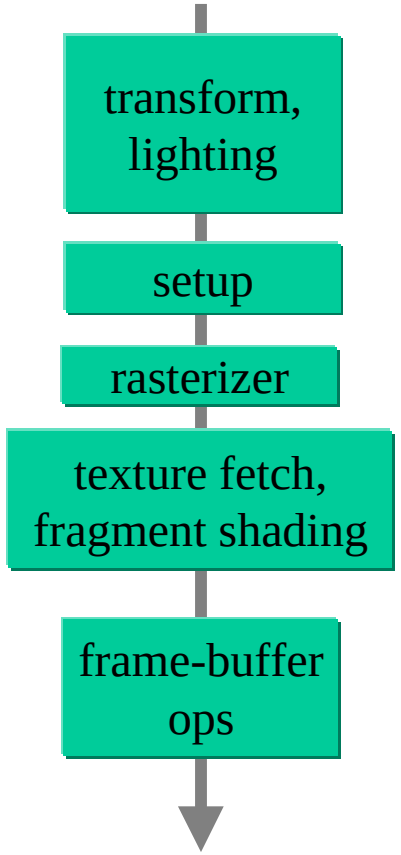
Vertex & fragment programs

- Code executed on the GPU
  - for each vertex (vertex program)
  - for each drawn pixel (fragment program)

- Must be « bound » like texture objects
- Only 1 vertex and 1 fragment program bound at once

ARB Vertex Programming Overview

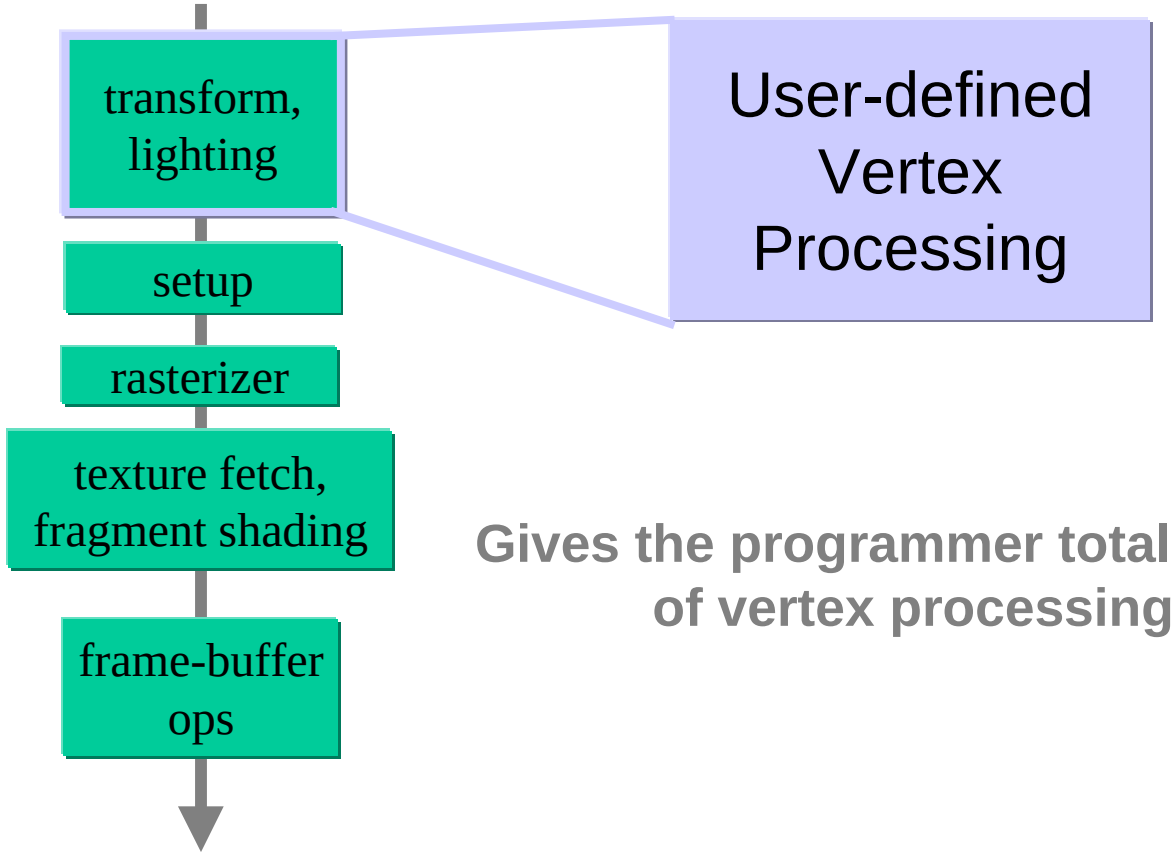
- Traditional Graphics Pipeline



Each unit has specific function (usually with configurable “modes” of operation)

ARB Vertex Programming Overview

- Vertex Programming offers programmable T&L unit



Gives the programmer total control of vertex processing.

What is Vertex Programming?

- Complete control of transform and lighting HW
- Complex vertex operations accelerated in HW
- Custom vertex lighting
- Custom skinning and blending
- Custom texture coordinate generation
- Custom texture matrix operations
- Custom vertex computations of your choice

- Offloading vertex computations frees up CPU

What is Vertex Programming?

- Vertex Program
  - Assembly language interface to T&L unit
  - GPU instruction set to perform all vertex math
  - Input: arbitrary vertex attributes
  - Output: a transformed vertex attributes
    - homogeneous clip space position (required)
    - colors (front/back, primary/secondary)
    - fog coord
    - texture coordinates
    - point size

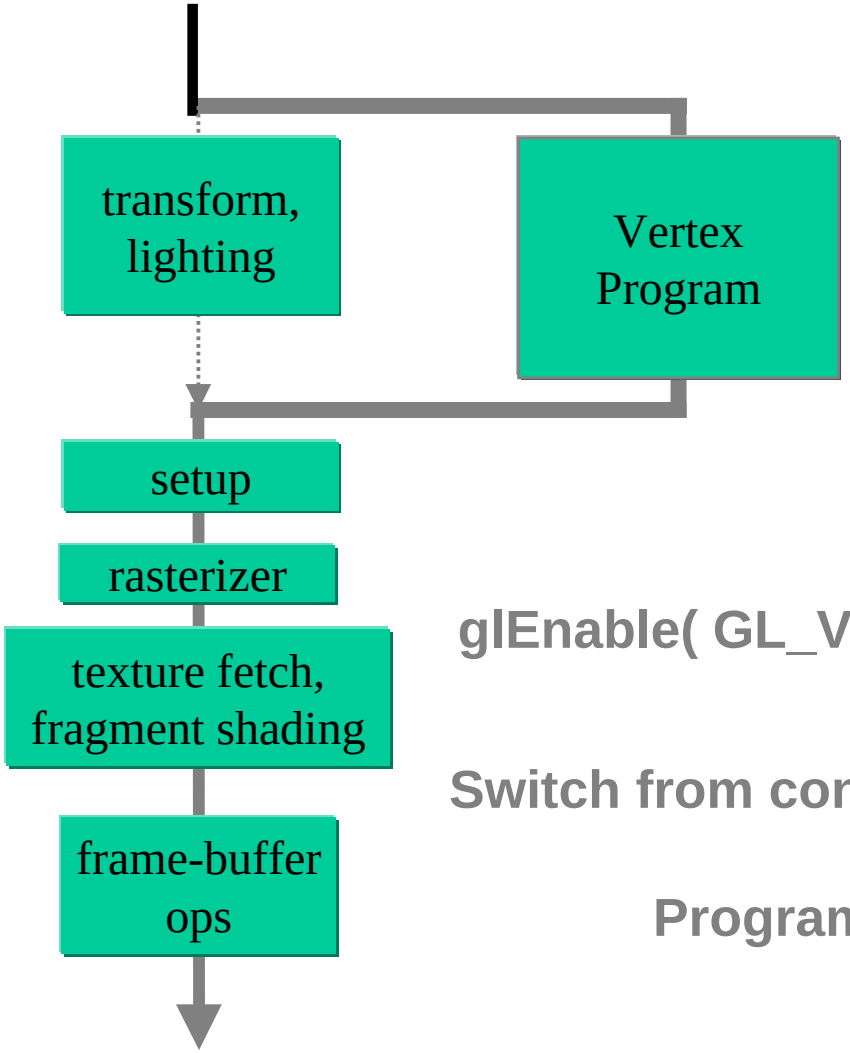
What is Vertex Programming?

- Vertex Program
  - Does not generate or destroy vertexes
    - 1 vertex in and 1 vertex out
  - No topological information provided
    - No edge, face, nor neighboring vertex info
  - Dynamically loadable
  - Exposed through NV_vertex_program and EXT_vertex_shader extensions
  - and now ARB_vertex_program

What is ARB_vertex_program?

- ARB_vertex_program is similar to NV_vertex_program with the addition of:
  - variables
  - local parameters
  - access to GL state
  - some extra instructions
  - implementation-specific resource limits

What is Vertex Programming?



glEnable(GL_VERTEX_PROGRAM_ARB);

Switch from conventional T&L model to Programmable mode

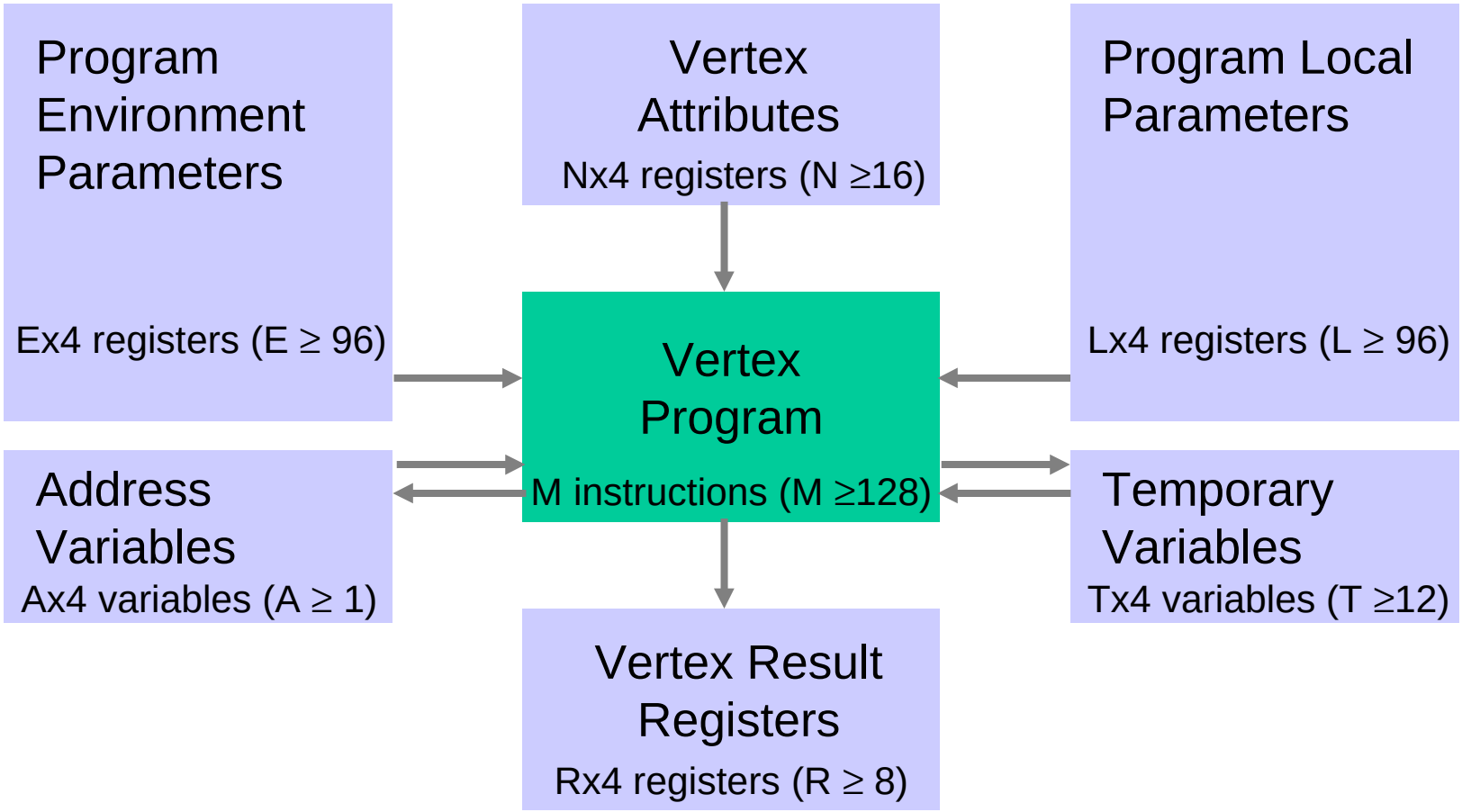
Specifically, what gets bypassed?

- **Modelview and projection vertex transformations**
- **Vertex weighting/blending**
- **Normal transformation, rescaling, normalization**
- **Color material**
- **Per-vertex lighting**
- **Texture coordinate generation and texture matrix transformations**
- **Per-vertex point size and fog coordinate computations**
- **User-clip planes**

What does NOT get bypassed?

- **Evaluators**
- **Clipping to the view frustum**
- **Perspective divide**
- **Viewport transformation**
- **Depth range transformation**
- **Front and back color selection (for two-sided)**
- **Clamping of primary and secondary colors to [0,1]**
- **Primitive assembly, setup, rasterization, blending**

Vertex Programming Conceptual Overview



Creating a Vertex Program

- **Programs are arrays of GLubytes (“strings”)**
- **Created/managed similar to texture objects**
  - **notion of a *program object***
  - **glGenProgramsARB(sizei n, uint *ids)**
  - **glBindProgramARB(enum target, uint id)**
  - **glProgramStringARB(enum target,**
    enum format,
    sizei len,
    const ubyte *program)

Creating a Vertex Program

```
Gluint progid;

// Generate a program object handle.
glGenProgramsARB( 1, &progid );

// Make the "current" program object progid.
glBindProgramARB( GL_VERTEX_PROGRAM_ARB, progid );

// Specify the program for the current object.
glProgramStringARB( GL_VERTEX_PROGRAM_ARB,
                    GL_PROGRAM_FORMAT_ASCII_ARB,
                    strlen(myString), myString );

// Check for errors and warnings...
```

Creating a Vertex Program

```
// Check for errors and warnings...
if ( GL_INVALID_OPERATION == glGetError() )
{
    // Find the error position
    GLint errPos;
    glGetIntergv( GL_PROGRAM_ERROR_POSITION_ARB,
                  &errPos );

    // Print implementation-dependent program
    // errors and warnings string.
    Glubyte *errString;
    glGetString( GL_PROGRAM_ERROR_STRING_ARB,
                 &errString );

    fprintf( stderr, "error at position: %d\n%s\n",
             errPos, errString );
}
```

Creating a Vertex Program

- **When finished with a program object, delete it**

```
// Delete the program object.
glDeleteProgramsARB( 1, &progid );
```

Specifying Program Parameters

- **Three types**
  - **Vertex Attributes – specifiable per-vertex**
  - **Program Local Parameters**
  - **Program Environment Parameters**

  Program Parameters modifiable outside of a Begin/End block

Specifying Vertex Attributes

- **Up to Nx4 per-vertex “generic” attributes**
- **Values specified with (several) new commands**

```
glVertexAttrib4fARB( index, x, y, z, w )
glVertexAttribs4fvARB( index, values )
```

- **Some entry points allow component-wise linear re-mapping to [0,1] or [-1,1]**

```
glVertexAttrib4NubARB( index, x, y, z, w )
glVertexAttrib4NbvARB( index, values )
```

similar to `glColor4ub()` **and** `glColor4b()`

Specifying Vertex Attributes

Component-wise linear re-mapping

Suffix	Data Type	Min Value	Min Value Maps to
b	1-byte integer	-128	-1.0
s	2-byte integer	-32,768	-1.0
i	4-byte integer	-2,147,483,648	-1.0
ub	unsigned 1-byte integer	0	0.0
us	unsigned 2-byte integer	0	0.0
ui	unsigned 4-byte integer	0	0.0

Suffix	Data Type	Max Value	Max Value Maps to
b	1-byte integer	127	1.0
s	2-byte integer	32,767	1.0
i	4-byte integer	2,147,483,647	1.0
ub	unsigned 1-byte integer	255	1.0
us	unsigned 2-byte integer	65,535	1.0
ui	unsigned 4-byte integer	4,294,967,295	1.0

Specifying Vertex Attributes

- **Vertex Array support**
- `glVertexAttribPointerARB(`

```
          uint index,
          int size,
          enum type,
          boolean normalize,
          sizei stride,
          const void *pointer )
```

- **“normalize” flag indicates if values should be linearly remapped**

Specifying Vertex Attributes

- **Setting vertex attribute 0 provokes vertex program execution**
- **Setting any other vertex attribute updates the current values of the attribute register**

- **Conventional attributes may be specified with conventional per-vertex calls**
  - **glColor, glNormal, glWeightARB, etc.**
- **Not strict aliasing (like NV_vertex_program)**
  - **More on this later…**

Specifying Program Local Parameters

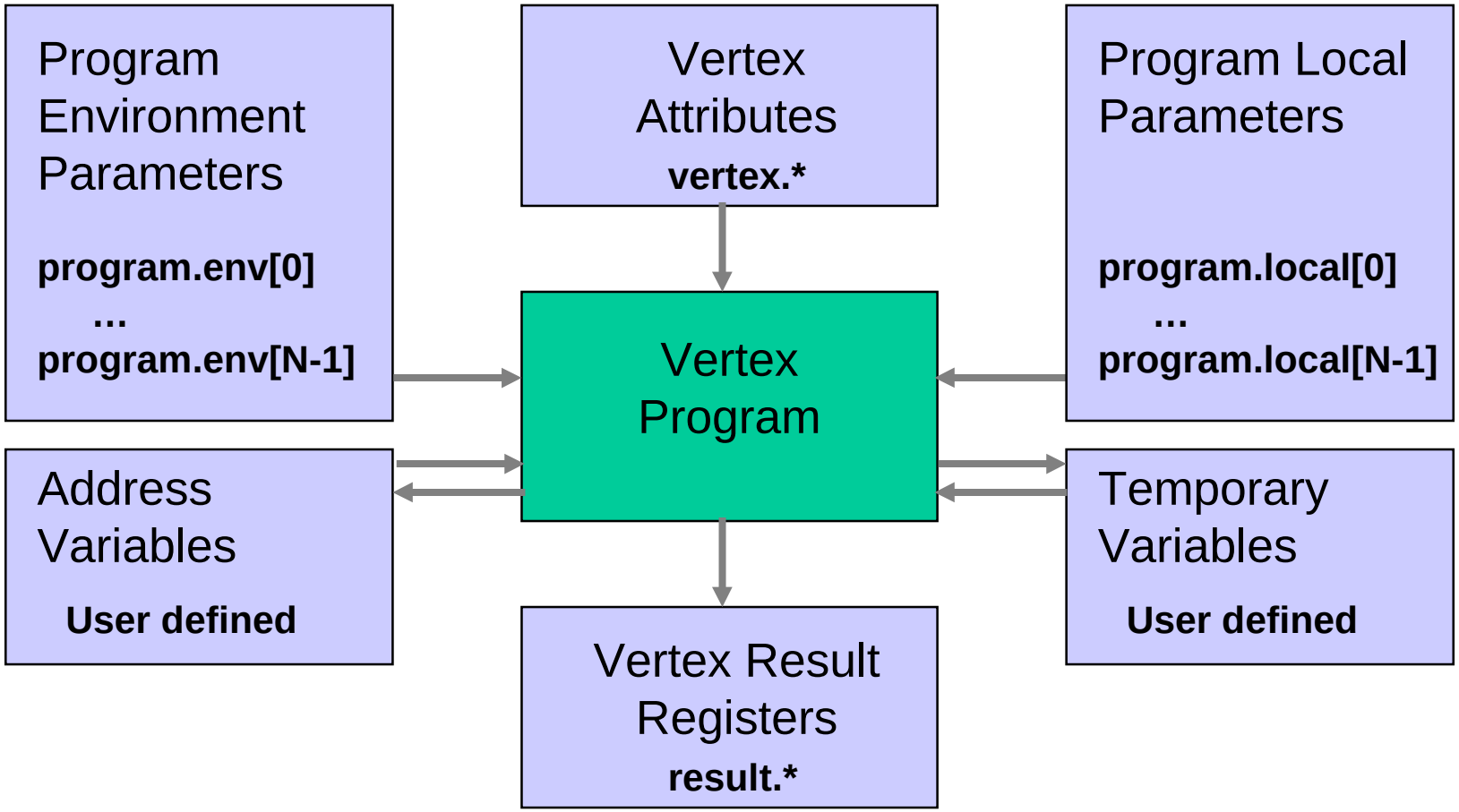
- **Each program object has an array of (N ≥ 96) four-component floating point vectors**
  - **Store program-specific parameters required by the program**
- **Values specified with new commands**
  - **glProgramLocalParameter4fARB(GL_VERTEX_PROGRAM_ARB, index, x, y, z, w)**
  - **glProgramLocalParameter4fvARB(GL_VERTEX_PROGRAM_ARB, index, params)**
- **Correspond to 96+ local parameter registers**

Specifying Program Environment Parameters

- **Shared array of (N ≥ 96) four-component registers accessible by any vertex program**
  - **Store parameters common to a set of program objects (i.e. Modelview matrix, MVP matrix)**

- **Values specified with new commands**
  - **glProgramEnvParameter4fARB(GL_VERTEX_PROGRAM_ARB, index, x, y, z, w)**
  - **glProgramEnvParameter4fvARB(GL_VERTEX_PROGRAM_ARB, index, params)**
- **Correspond to 96+ environment registers**

The Register Set



Program Environment and Program Local Registers

- **Program environment registers**

  access using: `program.env[i]`

  i in [0,GL_MAX_PROGRAM_ENV_PARAMETERS_ARB-1]

- **Program local registers**

  access using: `program.local[i]`

  i in [0,GL_MAX_PROGRAM_LOCAL_PARAMETERS_ARB-1]

Vertex Attribute Registers

Attribute Register	Components	Underlying State
vertex.position	(x,y,z,w)	object position
vertex.weight	(w,w,w,w)	vertex weights 0-3
vertex.weight[n]	(w,w,w,w)	vertex weights n-n+3
vertex.normal	(x,y,z,1)	normal
vertex.color	(r,g,b,a)	primary color
vertex.color.primary	(r,g,b,a)	primary color
vertex.color.secondary	(r,g,b,a)	secondary color
vertex.fogcoord	(f,0,0,1)	fog coordinate
vertex.texcoord	(s,t,r,q)	texture coordinate, unit 0
vertex.texcoord[n]	(s,t,r,q)	texture coordinate, unit n
vertex.matrixindex	(i,i,i,i)	vertex matrix indices 0-3
vertex.matrixindex[n]	(i,i,i,i)	vertex matrix indices n-n+3
vertex.attrib[n]	(x,y,z,w)	generic vertex attribute n

Semantics defined by program, NOT parameter name

Vertex Result Registers

Result Register	Components	Description
result.position	(x,y,z,w)	position in clip coordinates
result.color	(r,g,b,a)	front-facing, primary color
result.color.primary	(r,g,b,a)	front-facing, primary color
result.color.secondary	(r,g,b,a)	front-facing, secondary color
result.color.front	(r,g,b,a)	front-facing, primary color
result.color.front.primary	(r,g,b,a)	front-facing, primary color
result.color.front.secondary	(r,g,b,a)	front-facing, secondary color
result.color.back	(r,g,b,a)	back-facing, primary color
result.color.back.primary	(r,g,b,a)	back-facing, primary color
result.color.back.secondary	(r,g,b,a)	back-facing, secondary color
result.fogcoord	(f,*,*,*)	fog coordinate
result.pointsize	(s,*,*,*)	point size
result.texcoord	(s,t,r,q)	texture coordinate, unit 0
result.texcoord[n]	(s,t,r,q)	texture coordinate, unit n

Semantics defined by down-stream pipeline stages

Address Register Variables

- four-component signed integer vectors where only the ‘x’ component is addressable.

- Must be “declared” before use – address register variables

```
ADDRESS Areg;
ADDRESS A0;
ADDRESS A1, Areg;
```

- Number of variables limited to GL_MAX_PROGRAM_ADDRESS_REGISTERS_ARB

Temporary Variables

- Four-component floating-point vectors used to store intermediate computations
- Temporary variables declared before first use

```
TEMP flag;
TEMP tmp, ndotl, keenval;
```

- Number of temporary variables limited to GL_MAX_PROGRAM_TEMPORARIES_ARB

Identifiers and Variable Names

- **Any sequence of one or more**
  - **letters (A to Z, a to z),**
  - **digits (“0” to ”9”)**
  - **underscores (“_”)**
  - **dollar signs “$”**
- **First character may not be a digit**
- **Case sensitive**
- **Legal:** **A, b, _ab, $_ab, a$b, $_**
- **Not Legal:** **9A, ADDRESS, TEMP (other reserved words)**

Program Constants

- **Floating-point constants may be used in programs**
- **Standard format**

<integer portion> . <fraction portion> {“e”<integer>| “E”<integer>}

One (not both) may be omitted
Decimal or exponent (not both) may be omitted

- **Some Legal examples**

4.3, 4., .3, 4.3e3, 4.3e-3, 4.e3, 4e3, 4.e-3, .3e3

Program Parameter Variables

- **Set of four-component floating point vectors used as constants during program execution**
- **May be single four-vector or array of four-vectors**
- **Bound either**
  - **Explicitly (declaration of “param” variables)**
  - **Implicitly (inline usage of constants)**

Program Parameter Variable Bindings

- **Explicit Constant Binding**
- **Single Declaration**

```
PARAM a = {1.0, 2.0, 3.0, 4.0};   (1.0, 2.0, 3.0, 4.0)
PARAM b = {3.0};                  (3.0, 0.0, 0.0, 1.0)
PARAM c = {1.0, 2.0};             (1.0, 2.0, 0.0, 1.0)
PARAM d = {1.0, 2.0, 3.0 };       (1.0, 2.0, 3.0, 1.0)
PARAM e = 3.0;                    (3.0, 3.0, 3.0, 3.0)
```

- **Array Declaration**

```
PARAM arr[2] = { {1.0, 2.0, 3.0, 4.0},
                 {5.0, 6.0, 7.0, 8.0} };
```

Program Parameter Variable Bindings

- **Implicit Constant Binding**

`ADD a, b, {1.0, 2.0, 3.0, 4.0};`	**(1.0, 2.0, 3.0, 4.0)**
`ADD a, b, {3.0};`	**(3.0, 0.0, 0.0, 1.0)**
`ADD a, b, {1.0, 2.0};`	**(1.0, 2.0, 0.0, 1.0)**
`ADD a, b, {1.0, 2.0, 3.0};`	**(1.0, 2.0, 3.0, 1.0)**
`ADD a, b, 3.0;`	**(3.0, 3.0, 3.0, 3.0)**

- **Number of program parameter variables (explicit+implicit) limited to GL_MAX_PROGRAM_PARAMETERS_ARB**

Program Parameter Variable Bindings

- **Program Environment/Local Parameter Binding**

```
PARAM a = program.local[8];
PARAM b = program.env[9];
PARAM arr[2] = program.local[4..5];
PARAM mat[4] = program.env[0..3];
```

- **Essentially creates a “Reference”**

Program Parameter Variable Bindings

- **Material Property Binding**
  - **Bind to current GL material properties**

```
PARAM ambient = state.material.ambient;
PARAM diffuse = state.material.diffuse;
```

- **Additional material state to bind to…**

Program Parameter Variable Bindings

Binding	Components	Underlying GL state
state.material.ambient	(r,g,b,a)	front ambient material color
state.material.diffuse	(r,g,b,a)	front diffuse material color
state.material.specular	(r,g,b,a)	front specular material color
state.material.emission	(r,g,b,a)	front emissive material color
state.material.shininess	(s,0,0,1)	front material shininess
state.material.front.ambient	(r,g,b,a)	front ambient material color
state.material.front.diffuse	(r,g,b,a)	front diffuse material color
state.material.front.specular	(r,g,b,a)	front specular material color
state.material.front.emission	(r,g,b,a)	front emissive material color
state.material.front.shininess	(s,0,0,1)	front material shininess
state.material.back.ambient	(r,g,b,a)	back ambient material color
state.material.back.diffuse	(r,g,b,a)	back diffuse material color
state.material.back.specular	(r,g,b,a)	back specular material color
state.material.back.emission	(r,g,b,a)	back emissive material color
state.material.back.shininess	(s,0,0,1)	back material shininess

Program Parameter Variable Bindings

- **Light Property Binding**

```
PARAM ambient = state.light[0].ambient;
PARAM diffuse = state.light[0].diffuse;
```

- **Additional light state to bind to…**
- **Also bind to**
  - **Texture coord generation state**
  - **Fog property state**
  - **Clip plane state**
  - **Matrix state**

Output Variables

- **Variables that are declared bound to any vertex result register**

```
OUTPUT ocol = result.color.primary;
OUTPUT opos = result.position;
```

- **Write-only, essentially a “reference”**

Aliasing of Variables

- **Allows multiple variable names to refer to a single underlying variable**

```
ALIAS var2 = var1;
```

- **Do not count against resource limits**

Additional Notes on Variables

- **May be declared anywhere prior to first usage**
- **ARB spec. details specific rules with regards to resource consumption**
  - **Rule of thumb – generally minimize/remove unessential variables to keep resource counts**
  - **Can always load a program then query resource counts if desired**

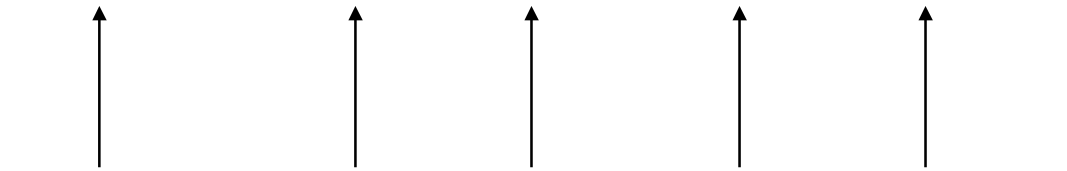
Vertex Programming Assembly Language

- **Powerful SIMD instruction set**
- **Four operations simultaneously**
- **27 instructions**
- **Operate on scalar or 4-vector input**
- **Result in a vector or replicated scalar output**

Assembly Language

Instruction Format:

Opcode dst, [-]s0 [,[-]s1 [,[-]s2]]; #comment



‘[’ and ‘]’ indicate optional modifiers

Examples:

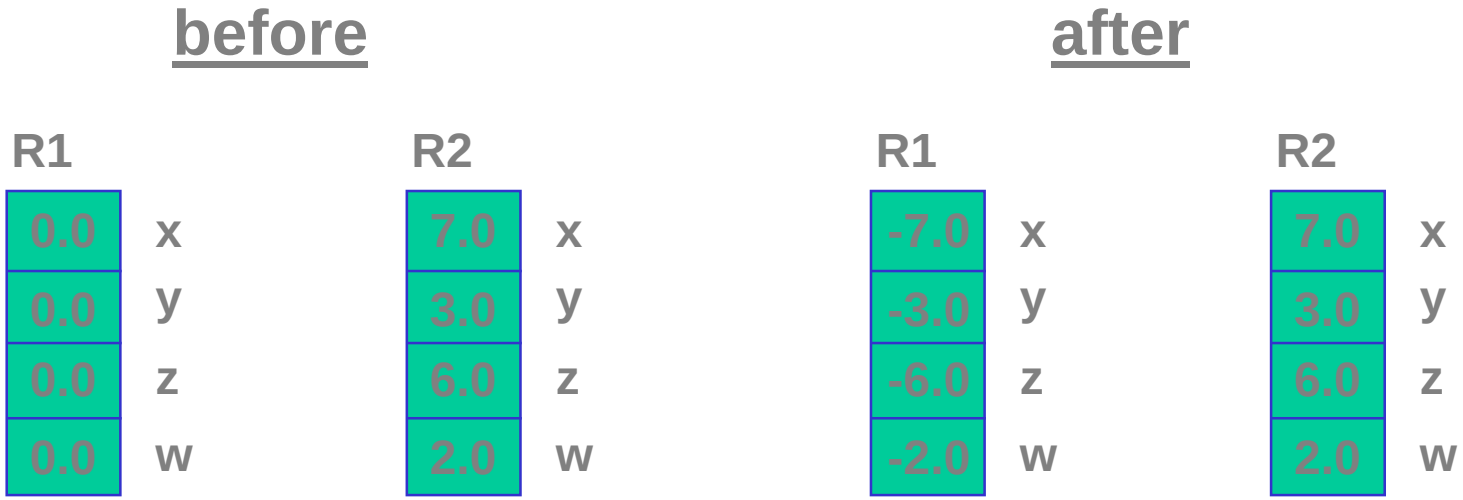
MOV R1, R2;

MAD R1, R2, R3, -R4;

Assembly Language

Source registers can be negated:

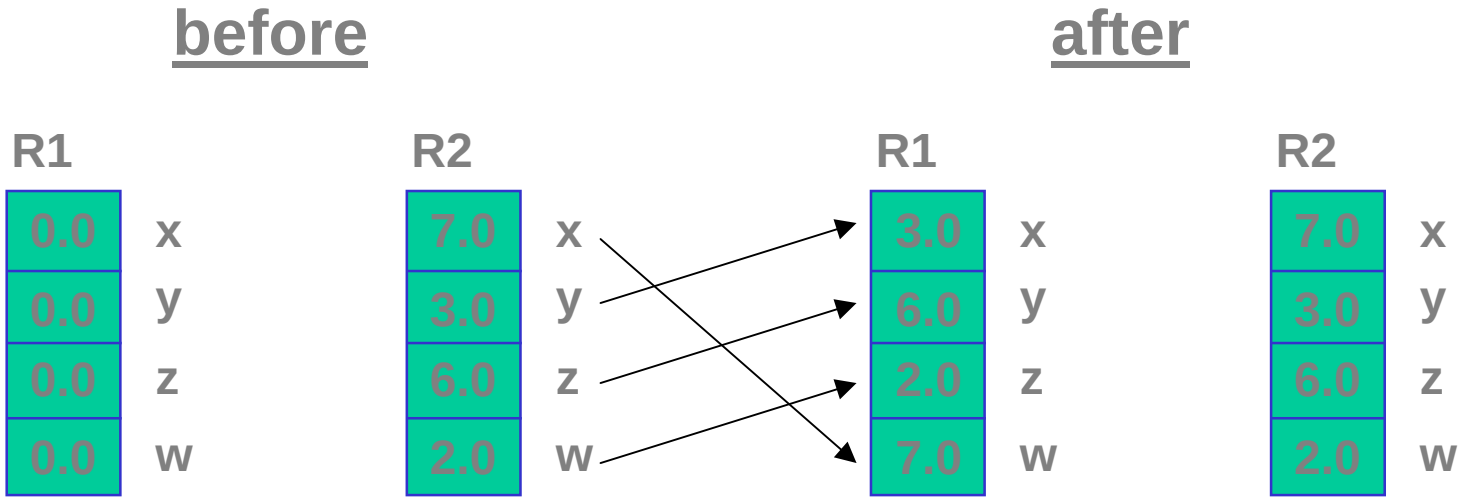
```
MOV     R1, -R2;
```



Assembly Language

Source registers can be “swizzled":

```
MOV     R1, R2.yzwx;
```



Note: MOV R1, R2.xxxx; ↔ MOV R1, R2.x;

Assembly Language

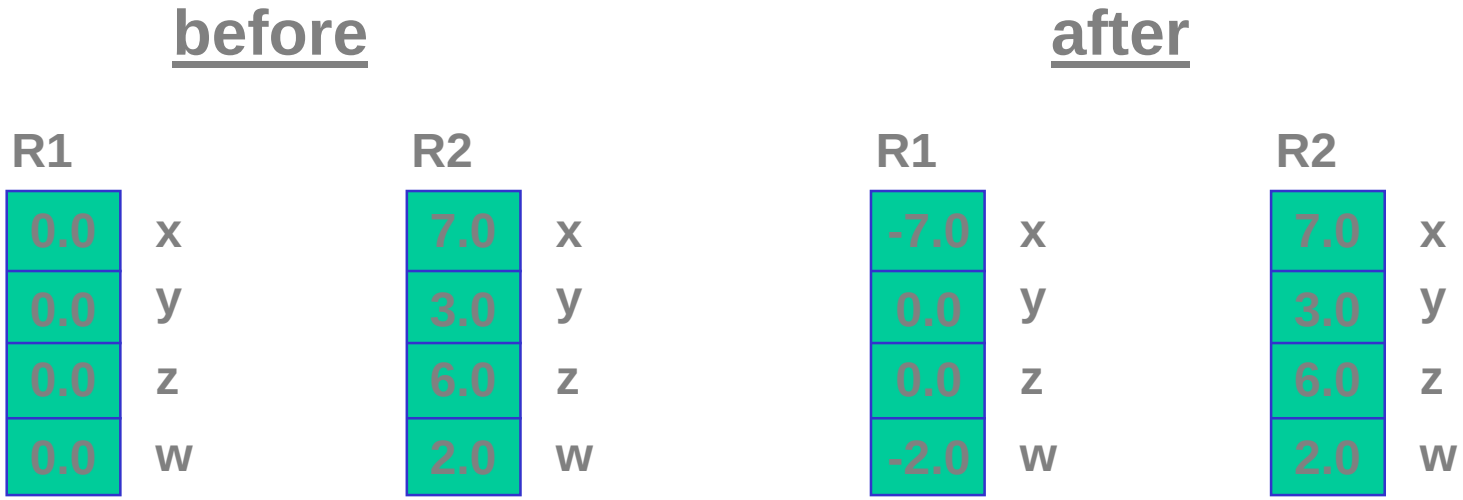
Destination register can mask which components are written to…

R1	$\Rightarrow$	**write all components**
R1.x	$\Rightarrow$	**write only x component**
R1.xw	$\Rightarrow$	**write only x, w components**

Vertex Programming Assembly Language

Destination register masking:

```
MOV     R1.xw, -R2;
```



Vertex Programming Assembly Language

There are 27 instructions in total …

- `ABS`
- `ADD`
- `ARL`
- `DP3`
- `DP4`
- `DPH`
- `DST`
- `EX2`
- `EXP`
- `FLR`
- `FRC`
- `LG2`
- `LIT`
- `LOG`
- `MAD`
- `MAX`
- `MIN`
- `MOV`
- `MUL`
- `POW`
- `RCP`
- `RSQ`
- `SGE`
- `SLT`
- `SUB`
- `SWZ`
- `XPD`

Example Program #1

Simple Transform to CLIP space

```
!!ARBvp1.0

ATTRIB pos = vertex.position;
PARAM  mat[4] = { state.matrix.mvp };

# Transform by concatenation of the
# MODELVIEW and PROJECTION matrices.
DP4    result.position.x, mat[0], pos;
DP4    result.position.y, mat[1], pos;
DP4    result.position.z, mat[2], pos;
DP4    result.position.w, mat[3], pos;

# Pass the primary color through w/o lighting.
MOV    result.color, vertex.color;

END
```

Example Program #2

Simple ambient, specular, and diffuse lighting (single, infinite light, local viewer)

```
!!ARBvp1.0

ATTRIB iPos      = vertex.position;
ATTRIB iNormal   = vertex.normal;
PARAM  mvinv[4] = { state.matrix.modelview.invtrans };
PARAM  mvp[4]    = { state.matrix.mvp };
PARAM  lightDir = state.light[0].position;
PARAM  halfDir  = state.light[0].half;
PARAM  specExp  = state.material.shininess;
PARAM  ambientCol  = state.lightprod[0].ambient;
PARAM  diffuseCol  = state.lightprod[0].diffuse;
PARAM  specularCol = state.lightprod[0].specular;
TEMP   eyeNormal, temp, dots, lightcoefs;
OUTPUT oPos      = result.position;
OUTPUT oColor    = result.color;
```

Example Program #2

```
# Transform the vertex to clip coordinates.
DP4     oPos.x, mvp[0], iPos;
DP4     oPos.y, mvp[1], iPos;
DP4     oPos.z, mvp[2], iPos;
DP4     oPos.w, mvp[3], iPos;

# Transform the normal into eye space.
DP3     eyeNormal.x, mvinv[0], iNormal;
DP3     eyeNormal.y, mvinv[1], iNormal;
DP3     eyeNormal.z, mvinv[2], iNormal;

# Compute diffuse and specular dot products
# and use LIT to compute lighting coefficients.
DP3     dots.x, eyeNormal, lightDir;
DP3     dots.y, eyeNormal, halfDir;
MOV     dots.w, specExp.x;
LIT     lightcoefs, dots;

# Accumulate color contributions.
MAD     temp, lightcoefs.y, diffuseCol, ambientCol;
MAD     oColor.xyz, lightcoefs.z, specularCol, temp;
MOV     oColor.w, diffuseCol.w;

END
```

Program Options

- OPTION mechanism for future extensibility
- Only one option: ARB_position_invariant
  - Guarantees position of vertex is same as what it would be if vertex program mode is disabled
  - User clipping also performed
  - Useful for “mixed-mode multi-pass”
- At start of program
  - `OPTION ARB_position_invariant`
- Error if program attempts to write to result.position

Querying Implementation-specific Limits

- **Max number of instructions**

```
glGetProgramivARB( GL_VERTEX_PROGRAM_ARB,
  GL_MAX_PROGRAM_INSTRUCTIONS, &maxInsts );
```

- **Max number of temporaries**

```
glGetProgramivARB( GL_VERTEX_PROGRAM_ARB,
  GL_MAX_PROGRAM_INSTRUCTIONS, &maxTemps );
```

- **Max number of program parameter bindings**

```
glGetProgramivARB( GL_VERTEX_PROGRAM_ARB,
  GL_MAX_PROGRAM_PARAMETERS, &maxParams );
```

- **Others (including native limits)**

Query current program resource usage by removing “MAX_”

Generic vs. Conventional Vertex Attributes

- **ARB_vertex_program spec allows for “fast and loose” storage requirements for generic and conventional attributes…**
- **Mapping between Generic Attributes and Conventional ones**

- **When a generic attribute is specified using glVertexAttrib*(), the current value for the corresponding conventional attribute becomes undefined**
  - **Also true for the converse**

Generic vs. Conventional Vertex Attributes

- **This allows implementations flexibility**
- **Mapping defined in the spec.**
- **Single programs may not access both**
  - **A generic attribute register**
  - **AND**
  - **Its corresponding conventional attribute register**
- **Error if it attempts to**

Generic and Conventional Attribute Mappings

Conventional Attribute	Generic Attribute
vertex.position	vertex.attrib[0]
vertex.weight	vertex.attrib[1]
vertex.weight[0]	vertex.attrib[1]
vertex.normal	vertex.attrib[2]
vertex.color	vertex.attrib[3]
vertex.color.primary	vertex.attrib[3]
vertex.color.secondary	vertex.attrib[4]
vertex.fogcoord	vertex.attrib[5]
vertex.texcoord	vertex.attrib[8]
vertex.texcoord[0]	vertex.attrib[8]
vertex.texcoord[1]	vertex.attrib[9]
vertex.texcoord[2]	vertex.attrib[10]
vertex.texcoord[3]	vertex.attrib[11]
vertex.texcoord[4]	vertex.attrib[12]
vertex.texcoord[5]	vertex.attrib[13]
vertex.texcoord[6]	vertex.attrib[14]
vertex.texcoord[7]	vertex.attrib[15]

In practice, probably use either conventional or generic not both

Introduction to ARB Fragment Programs

- New standardized programming model
- What it replaces
- What it does not replace

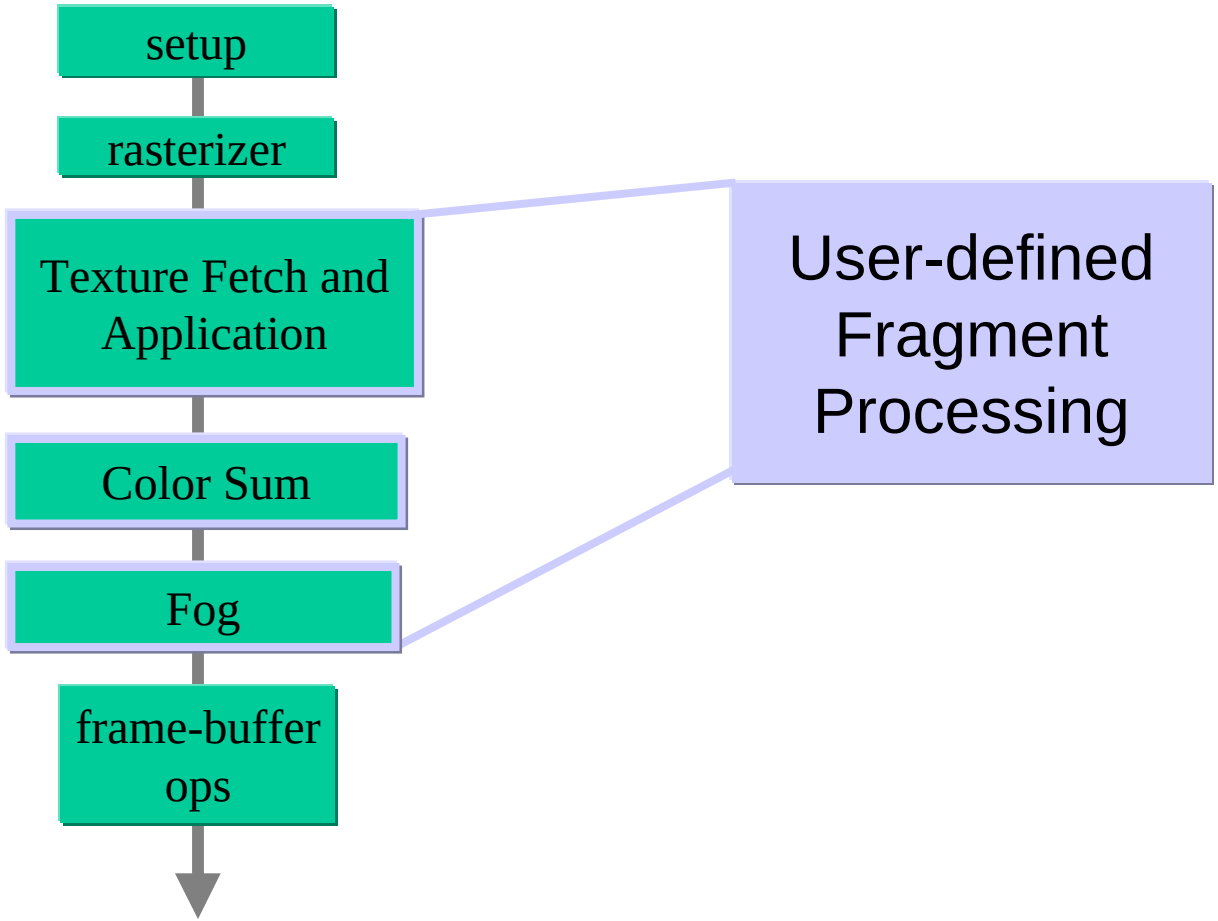
Standardized Fragment Programmability

- ARB standard for fragment-level programmability
- Derived from ARB_vertex_program

Replaces Multitexture Pipe

- Replaces
  - Texture blending
  - Color Sum
  - Fog
- Subsumes
  - texture enables
  - texture target priorities

Fragment Processing Pipe



Preserves Backend Operations

- Coverage application
- Alpha Test
- Stencil and depth test
  - try to process only visible pixels !!!
    - e.g., use a 1st pass to set-up z-buffer
- Blending

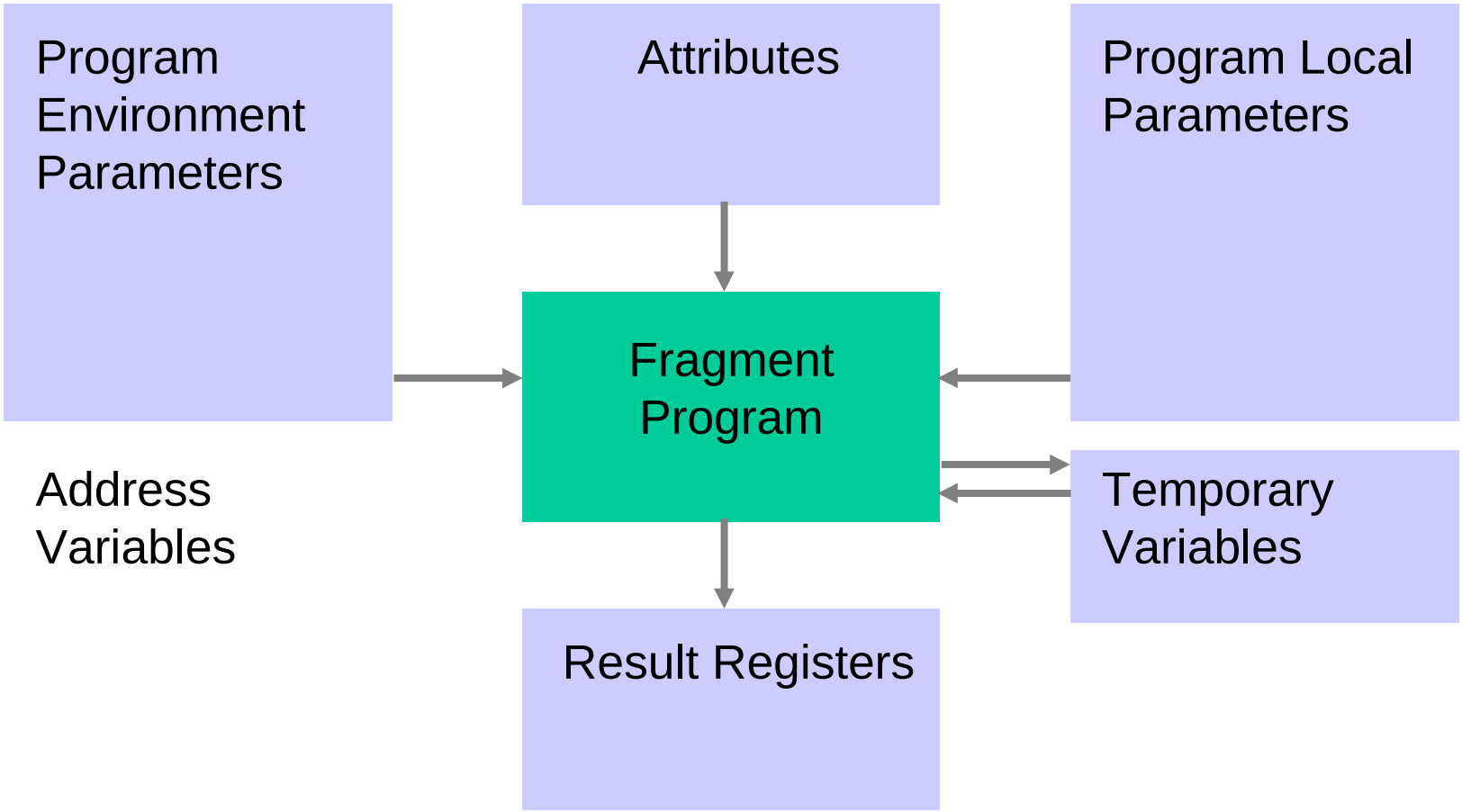
Programming Model

- ASM based similar to ARB_vertex_program
- Rich SIMD instruction set
- Requires resource management
  - Texture accesses
  - Interpolators
  - Temporaries
  - Constants

Similar to ARB_vertex_program

- Assembly syntax
- Same basic grammar rules
  - Ops are upper case
  - Statements terminated by semicolons
  - Comments begin with #

Fragment Program Machine



Large Instruction Set

- Three basic categories
  - Scalar (single value) operations
  - Vector operations
  - Texture operations
- Component swizzling
- Component masking

Scalar Instructions

- COS – cosine
- EX2 – base 2 exponential
- LG2 – base 2 logarithm
- RCP – reciprocal
- RSQ – reciprocal square root
- SIN – sine
- SCS – sine and cosine
- POW – power

New Scalar Instructions

- COS – cosine
- SIN – sine
- SCS – sine and cosine

Removed Scalar Instructions

- EXP – Partial Precision EX2
- LOG – Partial Precision LG2

Vector Instructions

- Standard Arithmetic
- Special purpose vector

Standard Arithmetic Ops

- ABS – absolute value
- FLR – floor
- FRC – fraction component
- SUB – subtract
- XPD – cross product
- CMP – compare
- LRP – linearly interpolate
- MAD – multiply accumulate
- MOV – move
- ADD – add
- DP3 – three component dot product
- DP4 – four component dot product

Special Vector Ops

- LIT – compute lighting
- DPH – homogeneous dot product
- DST – compute distance vector

New Instructions

- LRP – Linearly Interpolate
- CMP - Compare

Texture Instructions

- TEX
- TXP
- TXB
- KIL

Standard Texture Fetch

```
TEX <dest>, <src>, texture[n],
  <type>;
```

- Does not divide by q

Extended Texture Fetches

- TXP
  - Same syntax as TEX
  - Divides first three components by the fourth
- TXB
  - Adds the fourth component to the computed LOD

Killing pixels

- KIL instruction
  - Terminates shader program if any component is less than 0

Resource Categories in ARB FP

- Temporaries
- Textures
- Attributes
- Parameters
- Instructions
  - Texture Instructions
  - Arithmetic Instructions

Identifying Limits

- Standard Resource limits
  - Number of temps etc
- Texture Indirections

Using Fragment Program

- API
- Simple shaders
- Complex shaders

Simple API

- Loading programs
- Setting Parameters
- Making active

Shared API

```
glGenProgramsARB( num, id );

glBindProgramARB( GL_FRAGMENT_PROGRAM_ARB,
  id );

glProgramStringARB(
  GL_FRAGMENT_PROGRAM_ARB,
  GL_PROGRAM_FORMAT_ASCII_ARB, length,
  string );
```

Simple shaders

- Pre-programmable functionality
  - Texture * color

Simple Shader Example

```
!!ARBfp1.0

TEMP temp;   #temporary
ATTRIB tex0 = fragment.texcoord[0];
ATTRIB col0 = fragment.color;

PARAM pink = { 1.0, 0.4, 0.4, 1.0};

OUTPUT out = result.color;

TEX temp, tex0, texture[0], 2D;  #Fetch texture

MOV out, temp; #replace

#MUL out, col0, temp; #modulate
#MUL out, temp, pink; #modulate with constant color
```

Complex Shaders

- Lighting
- Procedural

Phong Lighting

```
#compute half angle vector
ADD spec.rgb, view, lVec;
DP3 spec.a, spec, spec;
RSQ spec.a, spec.a;
MUL spec.rgb, spec, spec.a;

#compute specular intensisty
DP3_SAT spec.a, spec, tmp;
LG2 spec.a, spec.a;
MUL spec.a, spec.a, const.w;
EX2 spec.a, spec.a;

#compute diffuse illum
DP3_SAT dif, tmp, lVec;
ADD_SAT dif.rgb, dif, const;
```

Procedural Bricks

```
#Apply the stagger
MUL r2.w, r0.y, half;
FRC r2.w, r2.w;
SGE r2.w, half, r2.w;
MAD r0.x, r2.w, half, r0.x;

#determine whether it is brick or mortar
FRC r0.xy, r0;
SGE r2.xy, freq, r0;
SUB r3.xy, 1.0, freq;
SGE r0.xy, r3, r0;
SUB r0.xy, r2, r0;
MUL r0.w, r0.x, r0.y;
```

Caveats

- Remember the difference between
  - {3.0}
  - 3.0
- Ensure Continuity at Boundaries
  - Needed to compute LOD
- Programs under limits may not load

High level programming: CG

- Syntax, operators, functions from C
- Conditionals and flow control
- Particularly suitable for GPUs:
  - Expresses data flow of the pipeline/stream architecture of GPUs (e.g. vertex-to-pixel)
  - Vector and matrix operations
  - Supports hardware data types for maximum performance
  - Exposes GPU functions for convenience and speed:
    - Intrinsic: (mul, dot, sqrt…)
    - Built-in: extremely useful and GPU optimized math, utility and geometric functions (noise, mix, reflect, sin…)
  - Compiler uses hardware profiles to subset Cg as required for particular hardware feature sets

Compiling Offline

At Development Time

Cg program source code

```
//
// Diffuse lighting
//
float d = dot(normalize(frag.N),
normalize(frag.L));
if (d < 0)
  d = 0;
c = d*tex2D(t, frag.uv)*diffuse;
...
```

Cg Compiler

Shader program assembly code

```
...
DP3 r0.x, f[TEX0], f[TEX0];
RSQ r0.x, r0.x;
MUL r0, r0.x, f[TEX0];
DP3 r1.x, f[TEX1], f[TEX1];
RSQ r1.x, r1.x;
MUL r1, r1.x, f[TEX1];
DP3 r0, r0, r1;
MAX r0.x, r0.x, 1.0;
MUL r0, r0.x, DIFFUSE;
TEX r1, f[TEX1], 0, 2D;
MUL r0, r0, r1;
...
```

Shader Compiler (nvasm.exe, psa.exe)

Shader program binary code

```
012b40  00 00 00 00 00 00 00 00
012b50  42 CD 09 84 51 3F 84 3C
012b60  93 AB D9 B1 87 87 70 B2
012b70  5C A5 D1 1C 58 65 58 F4
012b80  1F 27 1F 22 22 1F 1F 22
012b90  12 22 22 12 22 22 22 22
012ba0  1F 2F 2F 2F 2F 2F 23 FF
012bb0  37 37 37 37 37 37 2C 2C
012bc0  30 BE 47 04 4A BE A8 E6
012bd0  50 78 92 DD 90 B9 72 CE
012be0  03 69 8D EE 46 73 85 F9
```

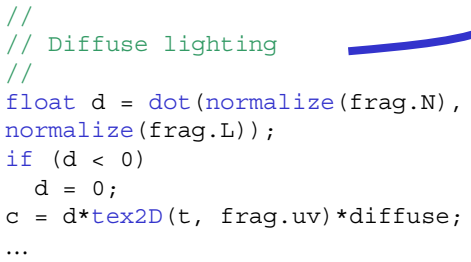
At Runtime

- At initialization:
  - Load assembly or binary program
- For every frame:
  - Load program parameters to hardware registers
  - Set rendering state
  - Load geometry
  - Render

Compiling at Runtime

At Development Time

Cg program source code



```
//
// Diffuse lighting
//
float d = dot(normalize(frag.N),
normalize(frag.L));
if (d < 0)
  d = 0;
c = d*tex2D(t, frag.uv)*diffuse;
...
```

At Runtime

- At initialization:
  - Compile and load Cg program
- For every frame:
  - Load program parameters with the Cg Runtime API
  - Set rendering state
  - Load geometry
  - Render

Pros and Cons of Runtime Compilation

- Pros:
  - Future compatibility: The application does not need to change to benefit from future compilers (future optimizations, future hardware)
  - Easy parameter management

- Cons:
  - Loading takes more time because of compilation
  - Cannot tweak the result of the compilation

NVIDIA Cg Usage

- Three ways:
- Cg Runtime – a thin API:
  - ASCII .Cg shader compiled at runtime
  - Convenient interface for setting shader parameters and constants
- Command line compiler generates text file output:
  - DX / OpenGL – vertex and pixel shader files
  - Tweak the ASM yourself
  - Generates comments on program params & registers
- CgFX
  - Effect framework with render states

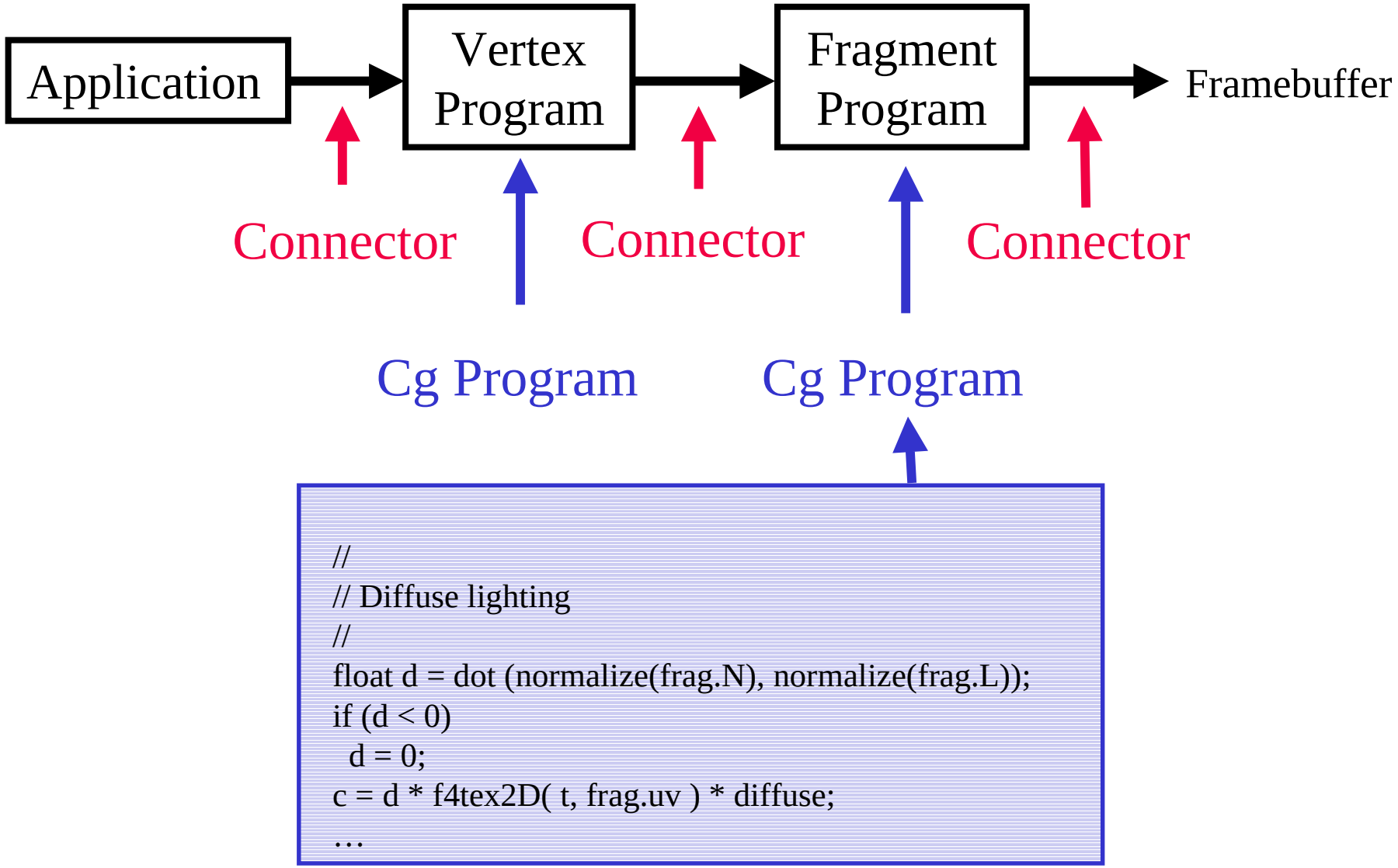
Flexible Adoption Path

- Can use system for just fragment programs
  - Define a connector to specify how you'll supply data to these programs
- Can use system for just vertex programs
  - Define connectors to specify how you'll supply data to these programs; and what they have to output.
- Can use system with older OpenGL applications
  - Classical `glVertex()`, `glNormal()` can still be used
  - Use OpenGL matrix tracking to provide modelview matrix to shading program.
  - But, must load program & supply light state

Mix and Match Any Method

Vertex processing	Fragment processing
Fixed function	Fixed function
-or-	-or-
Hand-written ASM	Hand-written ASM
-or-	-or-
Compiled Cg	Compiled Cg
-or-	-or-
Hand-optimized Cg ASM	Hand-optimized Cg ASM

Graphics Data Flow



Data types

- float = 32-bit IEEE floating point
- half = 16-bit IEEE-like floating point
- fixed = 12-bit fixed [-2,2) clamping (*OpenGL only*)
- bool = Boolean
- sampler* = Handle to a texture sampler

Array / vector / matrix declarations

- Declare vectors (up to length 4)
  and matrices (up to size 4x4)
  using built-in data types:

```
float4   mycolor;
float3x3 mymatrix;
```

- Declare more general arrays exactly as in C:

```
float lightpower[4];
```

- But, arrays are first-class types, not pointers
- Implementations may subset array
  capabilities to match HW restrictions

Extend standard arithmetic to vectors and matrices

- Component-wise `+ - * /` for vectors
- Dot product
  - `dot(v1,v2);` // returns a scalar
- Matrix multiplications:
  - assuming `float4x4 M` and `float4 v`
  - matrix-vector: `mul(M, v);` // returns a vector
  - vector-matrix: `mul(v, M);` // returns a vector
  - matrix-matrix: `mul(M, N);` // returns a matrix

New vector operators

- Swizzle operator extracts elements from vector

```
a = b.xxyy;
```

- Vector constructor builds vector

```
a = float4(1.0, 0.0, 0.0, 1.0);
```

“Profiles” for Specific HW Behavior

- Public NVIDIA Cg Compiler has three NV2X profiles:
  - DX8 Vertex Shader (vs1.1)
  - DX8 Pixel Shader (ps1.1)
  - OpenGL Vertex Program (currently based on NV_vertex_program, will move to ARB_vertex_program)
- Newest NVIDIA Cg Compiler currently has two NV30 profiles:
  - Vertex program (vp2.0)
  - Fragment Program (vp1.0)
- DX9 vertex/pixel shader profile support forthcoming
- Vertex profiles:
  - No “half” or “fixed” data type
  - No texture functions – It’s a vertex program!
- Fragment/pixel profiles:
  - No “for” or “while” loops (unless they’re unrollable)
  - etc.

Other profile limitations for NV30

- No pointers – not supported by HW
- Function parameters are passed by value/result
  - not by reference as in C++
  - use `out` or `inout` to declare output parameter
  - aliased parameters are written in order
- No unions or bit-fields
- No `int` data type

C++/Java like features

- Overloading

- Interfaces

```
// Declare interface to lights
interface Light {
    float3 direction(float3 from);
    float4 illuminate(float3 p, out float3 lv);
};
```

C++/Java like features

- An object that implement the interface

```
// Declare object type for point lights
struct PointLight : Light {
    float3 pos, color;
    float3 direction(float3 p) { return pos - p; }
    float3 illuminate(float3 p, out float3 lv) {
        lv = normalize(direction(p));
        return color;
    }
};
```

C++/Java-C# like features

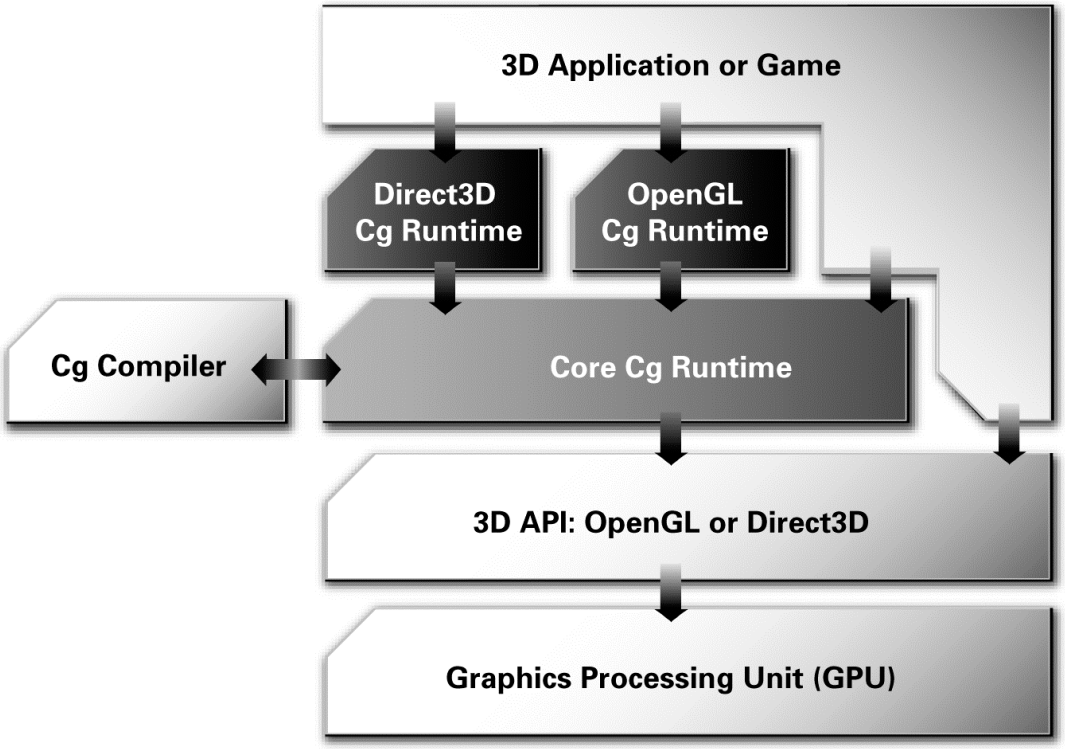
- Call object(s) via the interface type

```
// Main program (surface shader)
float4 main(appin IN, out float4 COUT,
                uniform Light lights[ ]) {
    ...
    for (int i=0; i < lights.Length; i++) { // for each light
        Cl = lights[i].illuminate(IN.pos, L); // get dir/color
        color += Cl * Plastic(texcolor, L, Nn, In, 30); // apply
        }
    COUT = color;
}
```

Cg Summary

- C-like language – expressive and efficient
- HW data types
- Vector and matrix operations
- Write separate vertex and fragment programs
- Connectors enable mix & match of programs by defining data flows
- Will be supported on any DX9 hardware
- Will support future HW (beyond NV30/DX9)

CG Runtime API



Core Cg Runtime

- Does *not* make any 3D API call
- Allows you to:
  - Create a context: `cgCreateContext()`
  - Compile a program for a given profile (vs_2_0, vs_2_x, ps_2_0, ps_2_x, arbvp1, arbvp1, vs_1_1, ps_1_1, fp20, etc...): `cgCreateProgram()`, `cgGetProgramString()`, etc...
  - Manage program parameters:
    - Iterate through the parameters: `cgGetFirstParameter()`, `cgGetNextParameter()`, etc...
    - Get parameter information: type, semantic, register, ...
  - Handle errors: `cgGetError()`, `cgSetErrorCallback()`, etc...

OpenGL Cg Runtime

- Makes the necessary OpenGL calls for you
- Allows you to:
  - Load a program into OpenGL: `cgGLLoadProgram()`
  - Enable a profile: `cgGLEnableProfile()`
  - Tell OpenGL to render with it: `cgGLBindProgram()`
  - Set parameter values:
    `cgGLSetParameter{1234}{fd}{v}()`,
    `cgGLSetParameterArray{1234}{fd}()`,
    `cgGLSetTextureParameter()`, etc...

Per pixel lighting

```
void main(float4 Pobject      : POSITION,
          float3 Nobject      : NORMAL,
          float2 TexUV        : TEXCOORD0,
          uniform float3 diffuse,
          uniform float3 specular,
          uniform float4x4 ModelViewProj,
          uniform float4x4 ModelView,
          uniform float4x4 ModelViewIT,

          out float4 HPosition    : POSITION,
          out float3 Peye         : TEXCOORD0,
          out float3 Neye         : TEXCOORD1,
          out float2 uv           : TEXCOORD2,
          out float3 Kd           : COLOR0,
          out float3 Ks           : COLOR1)
{
    // compute homogeneous position of vertex for rasterizer
    HPosition = mul(ModelViewProj, Pobject);

    Peye =  Pobject.xyz;
    Neye =  Nobject;

    // pass uv, Kd, and Ks through unchanged; if they are varying
    // per-vertex, however, they'll be interpolated before being
    // passed to the fragment program.
    uv = TexUV;
    Kd = diffuse;
    Ks = specular;
}
```

```
half diffuse(half4 l) { return l.y; }
half specular(half4 l) { return l.z; }


float4 main(float3 Peye         : TEXCOORD0,
            float3 Neye         : TEXCOORD1,
            float2 uv           : TEXCOORD2,
            float3 Kd           : COLOR0,
            float3 Ks           : COLOR1,
            uniform sampler2D diffuseMap,
            uniform float3 lightPos,
            uniform float shininess,
           uniform float3 eyePos) : COLOR
{
    // Normalize surface normal, vector to light source, and vector to the viewer
    float3 N = normalize(Neye);
    float3 L = lightPos - Peye;
    L = normalize(L);
    float3 V = normalize(eyePos-Peye);

    // Compute half-angle vector for specular lighting
    float3 H = normalize(L + V);

    float NdotL = dot(N, L), NdotH = dot(N, H);
    float4 lighting = lit(NdotL, NdotH, shininess);
    float3 C =  diffuse(lighting) * Kd * (float3)tex2D(diffuseMap, uv).xyz
                +  Ks*specular(lighting);

     return float4(C, 1);
}
```

Floating Point Textures/Pbuffers

- Vendor Specific Extensions
  - ATI_texture_float
  - NV_float_buffer
- Not filterable
  - Only NEAREST or NEAREST_MIPMAP_NEAREST

Floating Point Formats

- 32-bit
  - Essentially IEEE single precision
  - s23e8 format
- 16-bit
  - Reduced Range and precision
  - s10e5 format

Key Differences

- ATI
  - Extension Supports Filtering (SW)
  - All Standard Texture Formats
  - Available in Fixed Function and Programs
- NVIDIA
  - Rectangle Textures
    (u,v must be pre-multiplied by texture size !)
  - Vector-style Texture Formats
  - Available in Programs

Floating Point Targets

- Vendor Specific Extensions
  - NV_float_buffer
  - ATI_pixel_format_float
- Off-screen only
- Lack ‘Back End Operations’
  - No Coverage Application
  - No Alpha Test
  - No Alpha Blend
  - No Dither
  - No Logic Op

Key Differences

- NVIDIA
  - Available in Programs
  - Output skips ‘backend’ operations
  - Packs vectors
- ATI
  - Supports Fixed-Function and Programs
  - Most ‘backend’ operations still occur (SW)
  - Outputs separate vectors

Example: audio resampling, equalization & mixing

- Linear interpolation + 3 band attenuation + add
  - used for audio premixing in our clustering framework
- Audio data pre-stored in texture memory
  - unsigned 16-bit integer RGB
- Fragment program triggered by drawing 2D lines

Future

- glSLANG ?

- OpenGL ARB Superbuffers
  - dynamic memory used as frame buffer, texture, vertex array,…
- Faster PCI express bus
  - AGP x8 : 2.1 GB/ sec.
  - PCI express (x16) : 4 GB/sec. Bi-directionnel

References

- http://www.nvidia.com
- http://www.ati.com
- le livre « CG tutorial » est dispo à la doc
- http://www.cgshaders.org/
- http://www.gpgpu.org