University of British Columbia
CPSC 314 Computer Graphics
Jan-Apr 2007

Tamara Munzner

Blending, Modern Hardware

Week 12, Mon Apr 2

http://www.ugrad.cs.ubc.ca/~cs314/Vjan2007

Old News

- extra TA office hours in lab for hw/project Q&A
  - next week: Thu 4-6, Fri 10-2
  - last week of classes:
    - Mon 2-5, Tue 4-6, Wed 2-4, Thu 4-6, Fri 9-6
- final review Q&A session
  - Mon Apr 16 10-12
- reminder: no lecture/labs Fri 4/6, Mon 4/9

New News

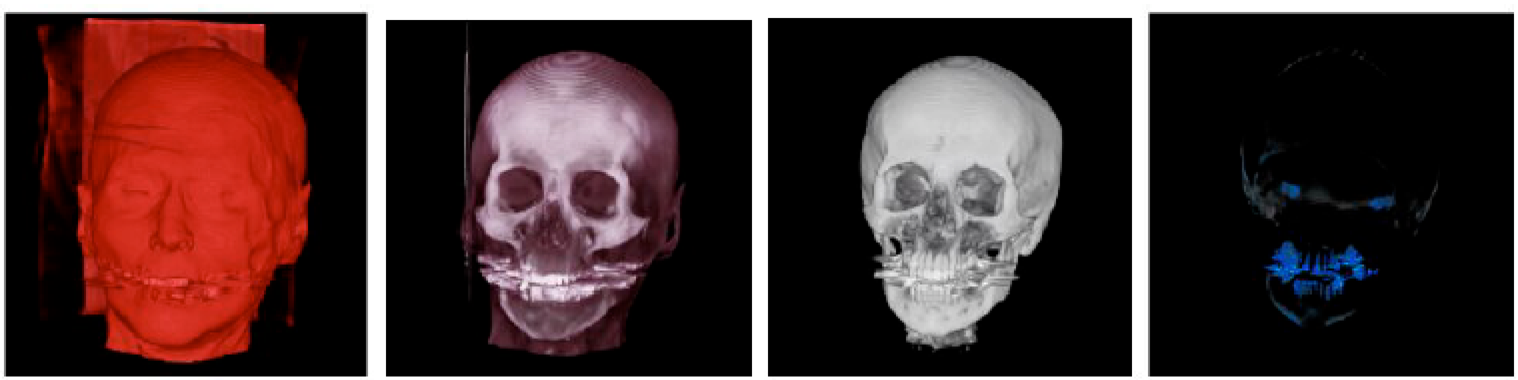
- project 4 grading slots signup
  - Wed Apr 18 10-12
  - Wed Apr 18 4-6
  - Fri Apr 20 10-1

Review: Volume Graphics

- for some data, difficult to create polygonal mesh
- voxels: discrete representation of 3D object
  - volume rendering: create 2D image from 3D object
- translate raw densities into colors and transparencies
  - different aspects of the dataset can be emphasized via changes in transfer functions

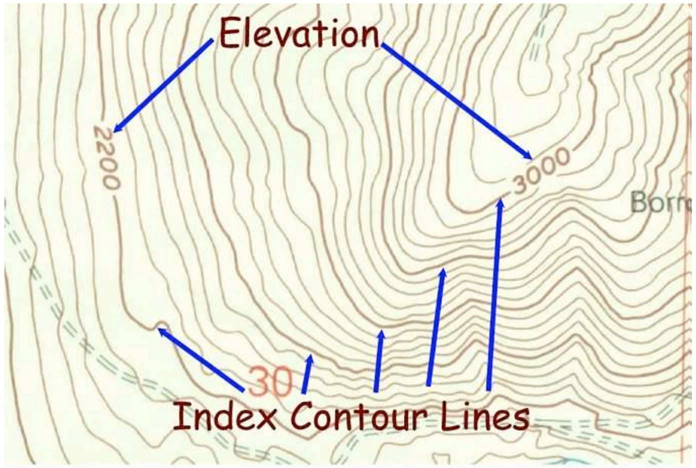
Review: Volume Graphics

- pros
  - formidable technique for data exploration
- cons
  - rendering algorithm has high complexity!
  - special purpose hardware costly (~$3K-$10K)

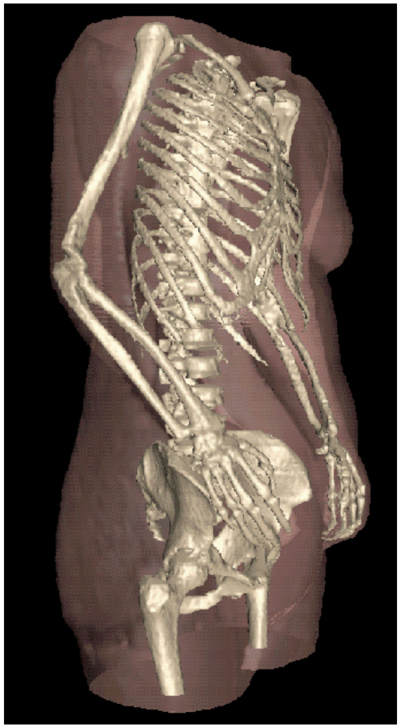
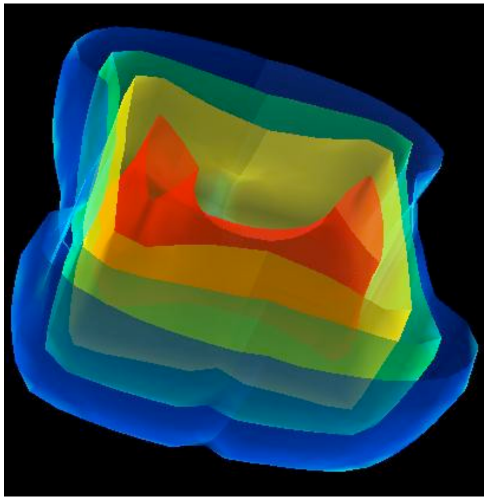
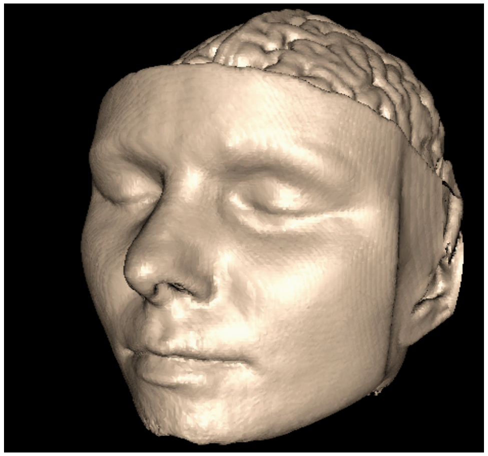


volumetric human head (CT scan)

Review: Isosurfaces

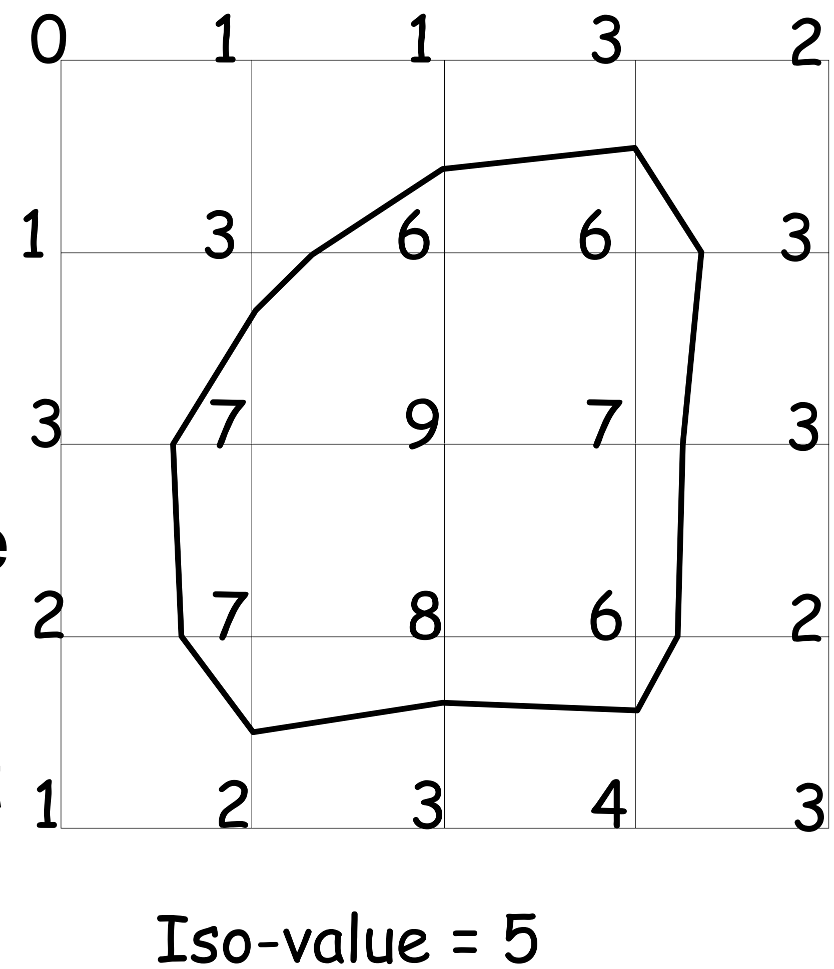


- 2D scalar fields: isolines
  - contour plots, level sets
  - topographic maps
- 3D scalar fields: isosurfaces

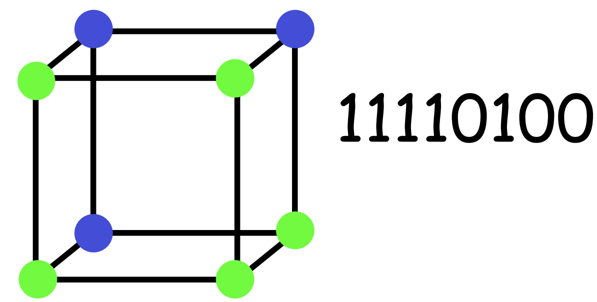
Review: Isosurface Extraction

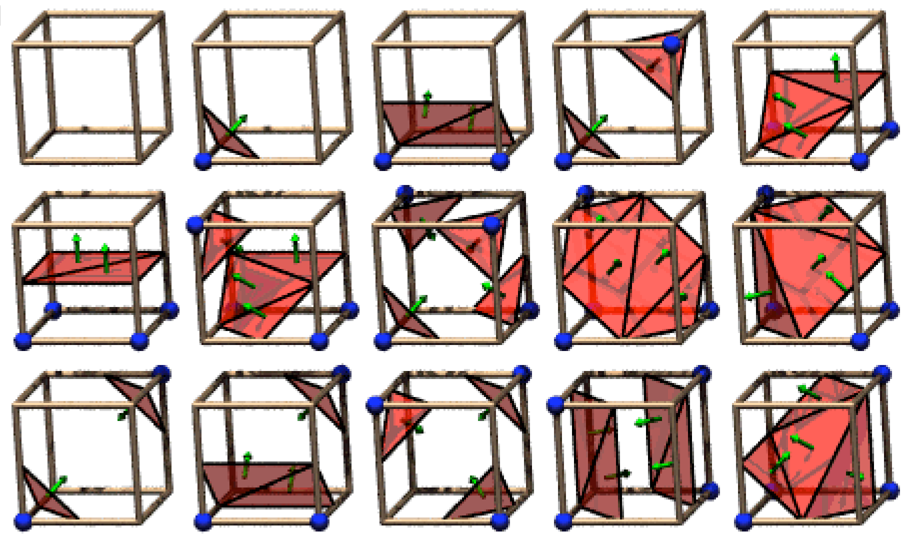
- array of discrete point samples at grid points
  - 3D array: voxels
- find contours
  - closed, continuous
  - determined by iso-value
- several methods
  - marching cubes is most common



Review: Marching Cubes

- create cube
- classify each voxel
- binary labeling of each voxel to create index
- use in array storing edge list
  - all 256 cases can be derived from 15 base cases
- interpolate triangle vertex
- calculate the normal at each cube vertex
- render by standard methods

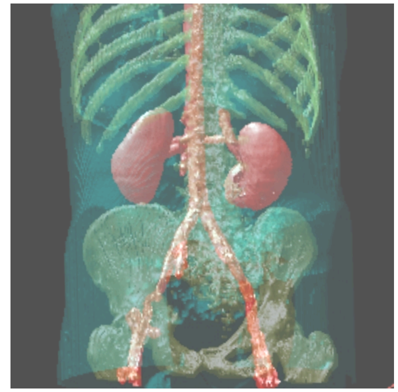


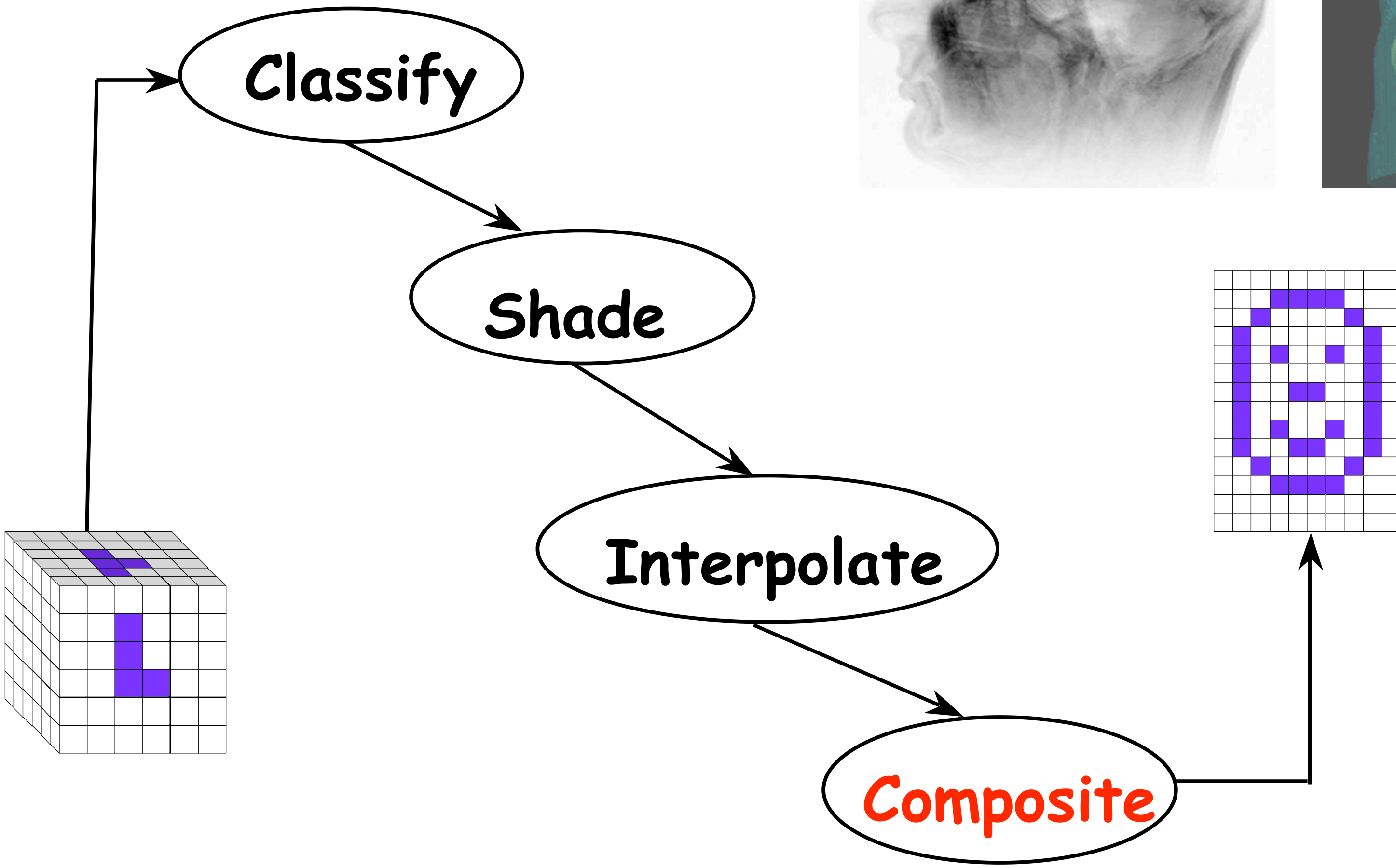


The 15 Cube Combinations

Review: Direct Volume Rendering Pipeline

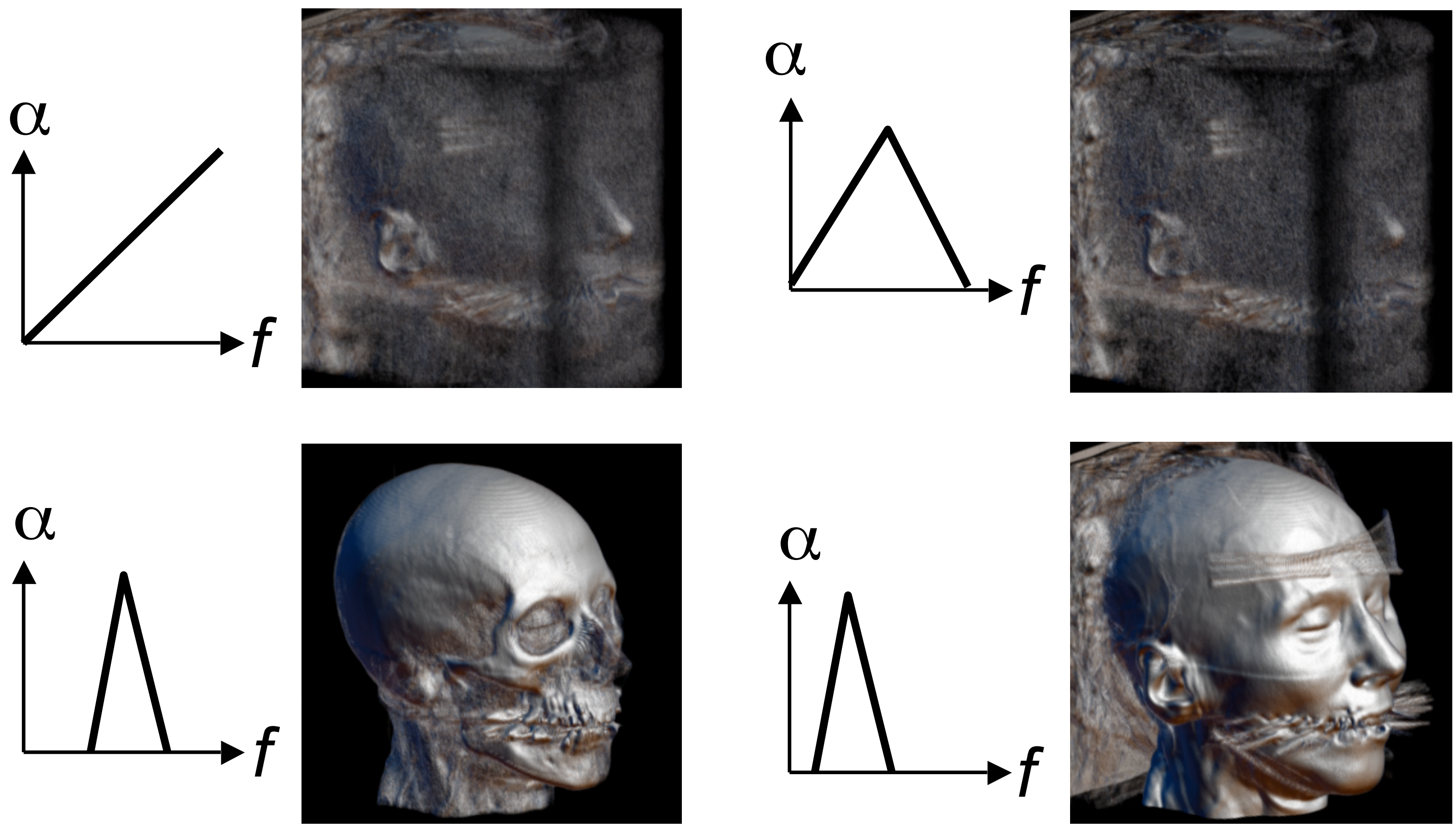
- do not compute surface





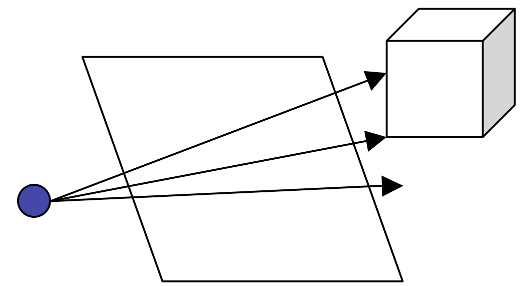
Review: Transfer Functions To Classify

- map data value to color and opacity
  - can be difficult, unintuitive, and slow

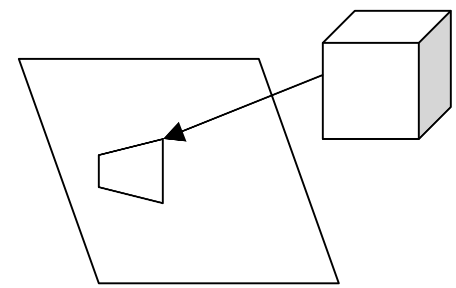


Review: Volume Rendering Algorithms

- ray casting
  - image order, forward viewing

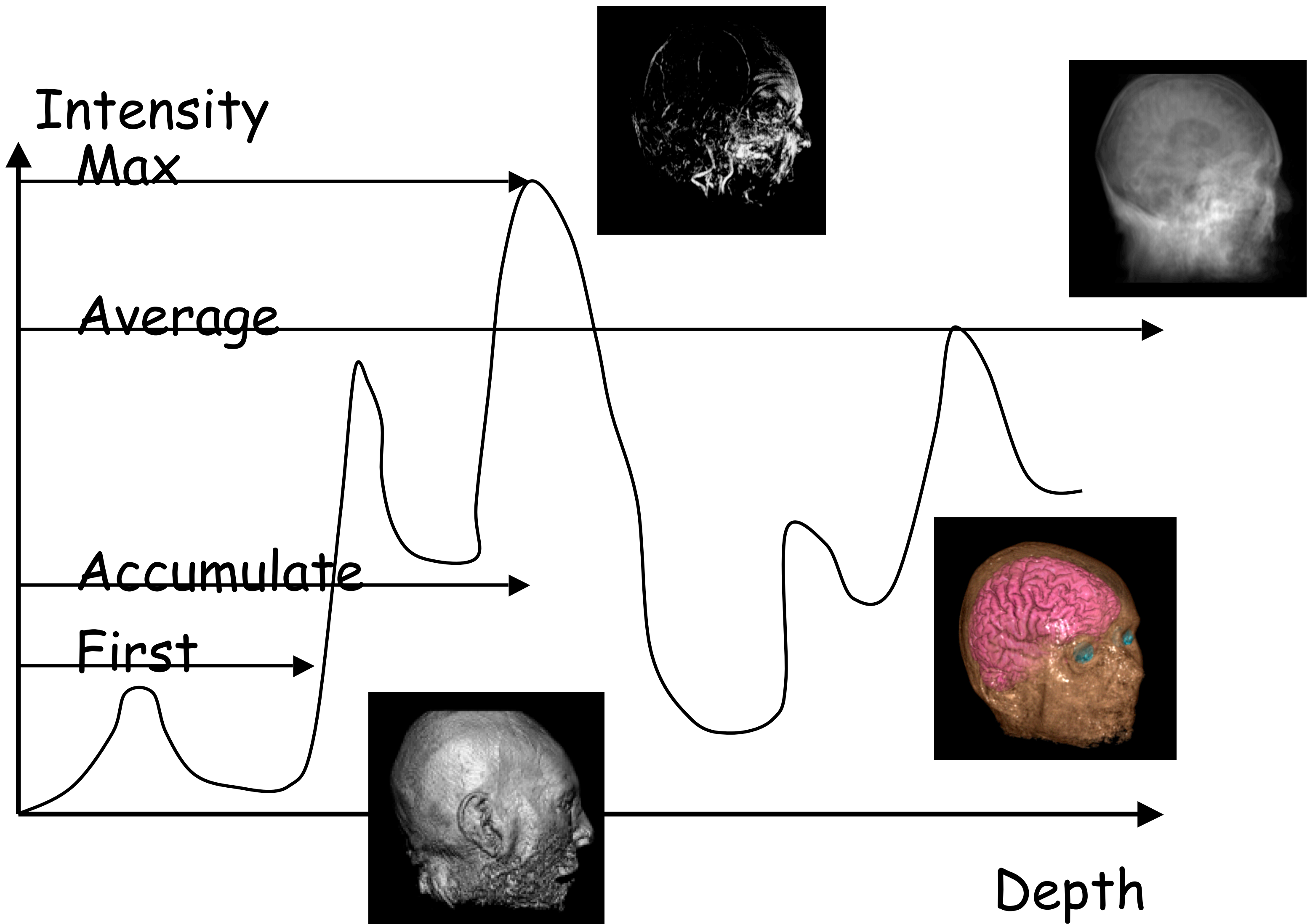


- splatting
  - object order, backward viewing



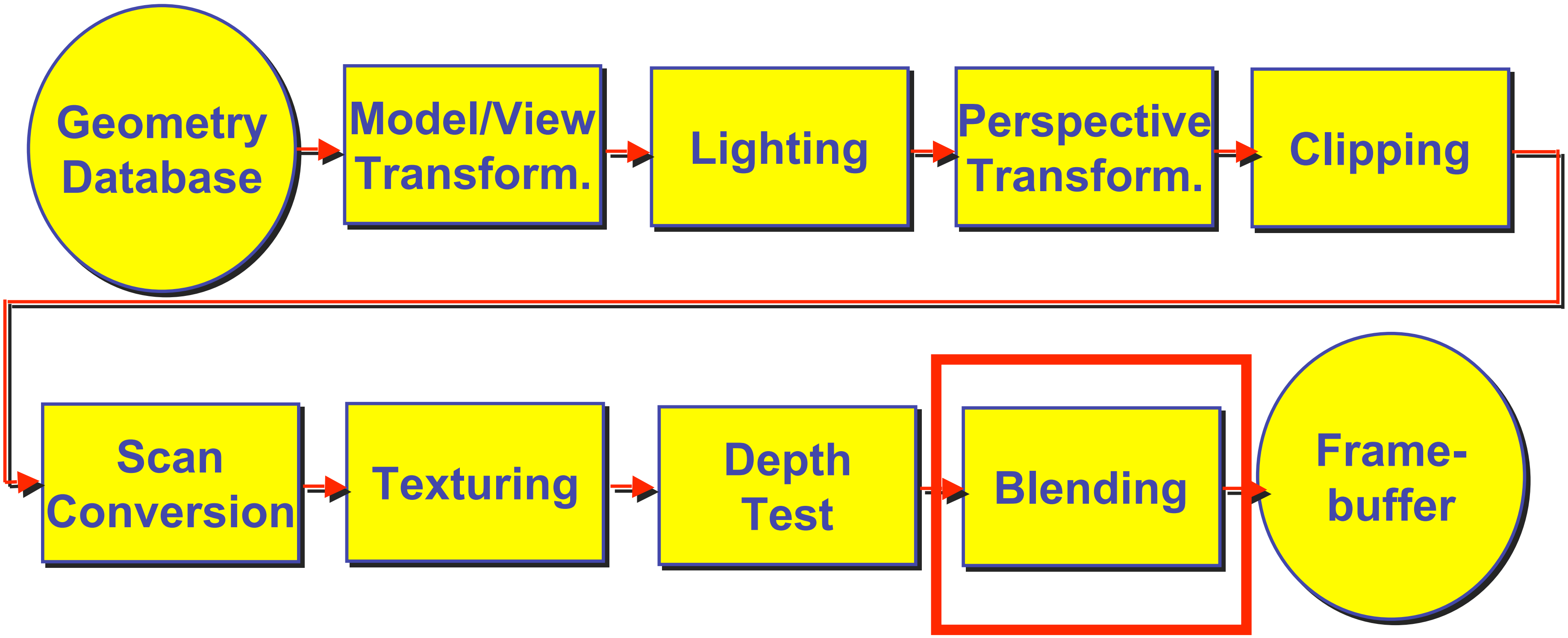
- texture mapping
  - object order
  - back-to-front compositing

Review: Ray Casting Traversal Schemes



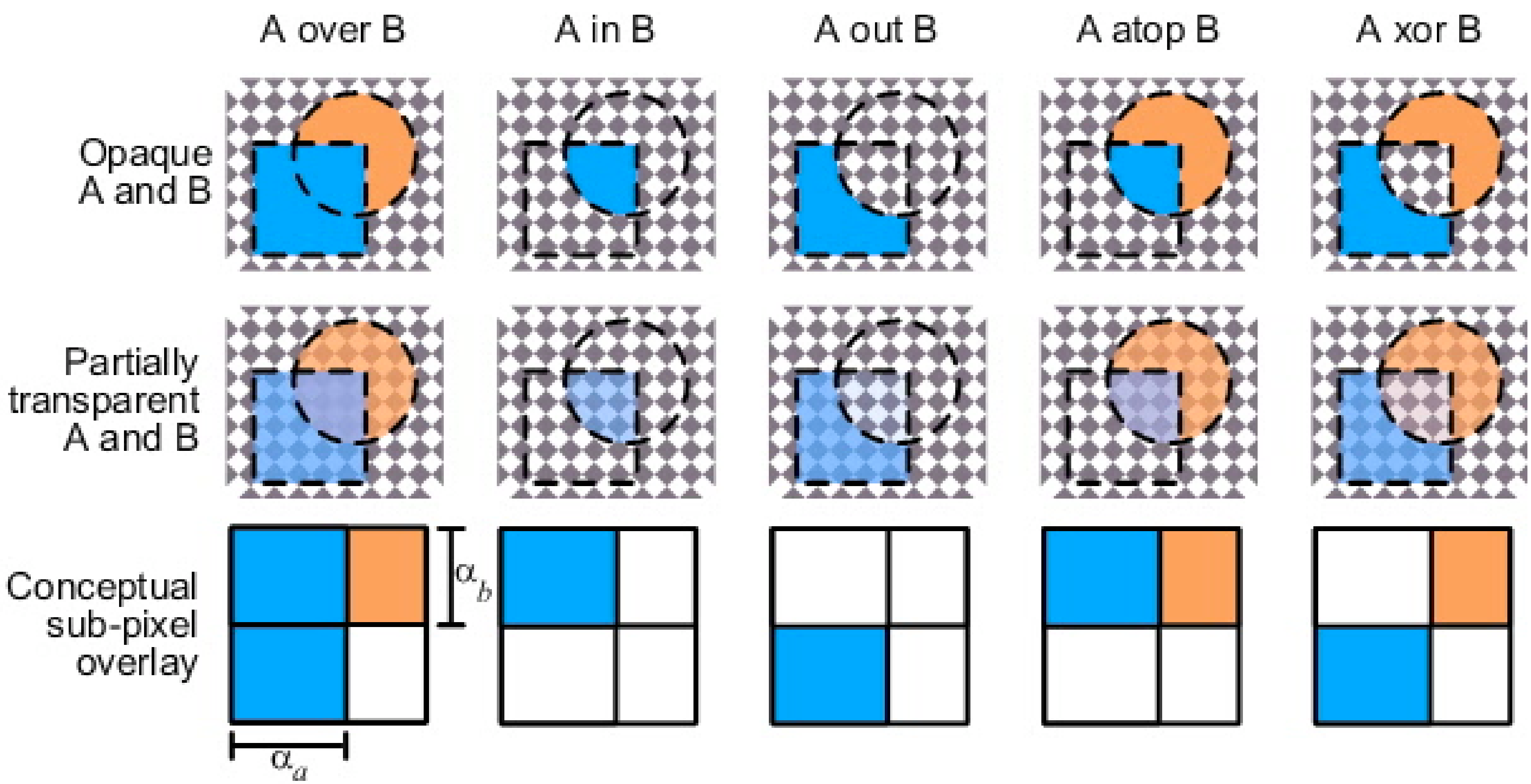
Blending

Rendering Pipeline



Blending/Compositing

- how might you combine multiple elements?
- foreground color **A**, background color **B**



Premultiplying Colors

- specify opacity with alpha channel: (r,g,b,α)
  - α=1: opaque, α=.5: translucent, α=0: transparent

- **A** *over* **B**
  - $\mathbf{C} = \alpha\mathbf{A} + (1-\alpha)\mathbf{B}$

- but what if **B** is also partially transparent**?**
  - $\mathbf{C} = \alpha\mathbf{A} + (1-\alpha)\ \beta\mathbf{B} = \ \beta\mathbf{B} + \alpha\mathbf{A} + \beta\mathbf{B} - \alpha\ \beta\mathbf{B}$
  - $\gamma = \beta + (1-\beta)\alpha = \beta + \alpha - \alpha\beta$
    - 3 multiplies, different equations for alpha vs. RGB

- premultiplying by alpha
  - $\mathbf{C'} = \gamma\ \mathbf{C},\ \mathbf{B'} = \beta\mathbf{B},\ \mathbf{A'} = \alpha\mathbf{A}$

  - $\mathbf{C'} = \mathbf{B'} + \mathbf{A'} - \alpha\mathbf{B'}$
  - $\gamma = \beta + \alpha - \alpha\beta$
    - 1 multiply to find C, same equations for alpha and RGB

Modern GPU Features

Reading

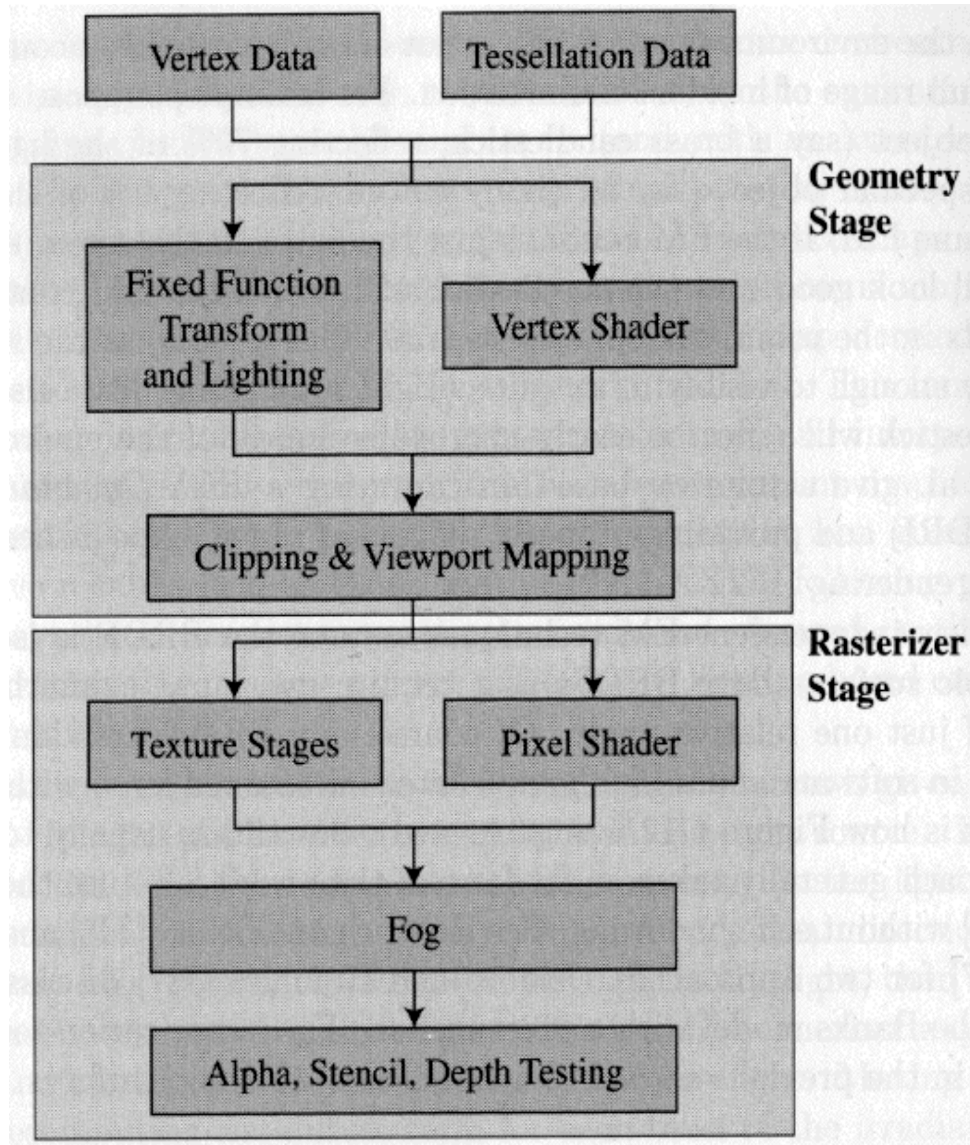
- FCG Chap17 Using Graphics Hardware
  - especially 17.3
- FCG Section 3.8 Image Capture and Storage

Rendering Pipeline

- so far
  - rendering pipeline as a specific set of stages with **fixed functionality**
- modern graphics hardware more flexible
  - programmable “vertex shaders” replace several geometry processing stages
  - programmable “fragment/pixel shaders” replace texture mapping stage
  - hardware with these features now called Graphics Processing Unit (GPU)

Modified Pipeline

- vertex shader
  - replaces model/view, lighting, and perspective
  - have to implement these yourself
  - but can also implement much more
- fragment/pixel shader
  - replaces texture mapping
  - fragment shader must do texturing
  - but can do other things



Vertex Shader Motivation

- hardware transform and lighting:
  - i.e. hardware geometry processing
  - was mandated by need for higher performance in the late 90s
  - previously, geometry processing was done on CPU, except for very high end machines
  - downside: now limited functionality due to fixed function hardware

Vertex Shaders

- programmability required for more complicated effects
  - tasks that come before transformation vary widely
  - putting every possible lighting equation in hardware is impractical
  - implementing programmable hardware has advantages over CPU implementations
    - better performance due to massively parallel implementations
    - lower bandwidth requirements (geometry can be cached on GPU)

Vertex Program Properties

- run for every vertex, independently
  - access to all per-vertex properties
    - position, color, normal, texture coords, other custom properties
  - access to read/write registers for temporary results
    - value is reset for every vertex
    - cannot pass information from one vertex to the next
  - access to read-only registers
    - global variables like light position, transformation matrices
  - write output to a specific register for resulting color

Vertex Shaders/Programs

- concept
  - programmable pipeline stage
    - floating-point operations on 4 vectors
      - points, vectors, and colors!
  - replace all of
    - model/view transformation
    - lighting
    - perspective projection

Vertex Shaders/Programs

- a little assembly-style program is executed on every individual vertex
- it sees:
  - vertex attributes that change per vertex:
    - position, color, texture coordinates…
  - registers that are constant for all vertices (changes are expensive):
    - matrices, light position and color, …
  - temporary registers
  - output registers for position, color, tex coords…

Vertex Programs Instruction Set

- arithmetic operations on 4-vectors:
  - ADD, MUL, MAD, MIN, MAX, DP3, DP4
- operations on scalars
  - RCP (1/x), RSQ (1/√x), EXP, LOG
- specialty instructions
  - DST (distance: computes length of vector)
  - LIT (quadratic falloff term for lighting)

- very latest generation:
  - loops and conditional jumps
  - still more expensive than straightline code

Vertex Programs Applications

- what can they be used for?
  - can implement all of the stages they replace
  - but can allocate resources more dynamically
    - e.g. transforming a vector by a matrix requires 4 dot products
    - enough memory for 24 matrices
    - can arbitrarily deform objects
      - procedural freeform deformations
    - lots of other applications
      - shading
      - refraction
      - …

Skinning

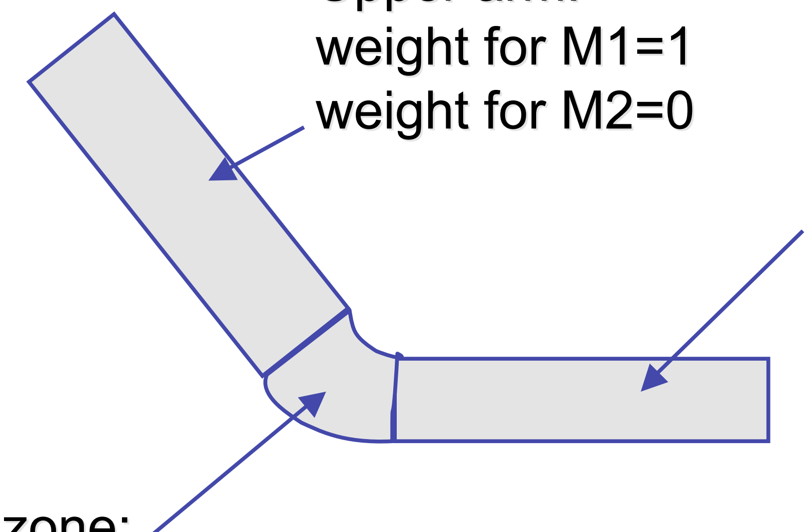
- want to have natural looking joints on human and animal limbs
- requires deforming geometry, e.g.
  - single triangle mesh modeling both upper and lower arm
  - if arm is bent, upper and lower arm remain more or less in the same shape, but transition zone at elbow joint needs to deform

Skinning

- approach:
  - multiple transformation matrices
    - more than one model/view matrix stack, e.g.
      - one for model/view matrix for lower arm, and
      - one for model/view matrix for upper arm
    - every vertex is transformed by <u>both</u> matrices
      - yields 2 different transformed vertex positions!
    - use per-vertex blending weights to interpolate between the two positions

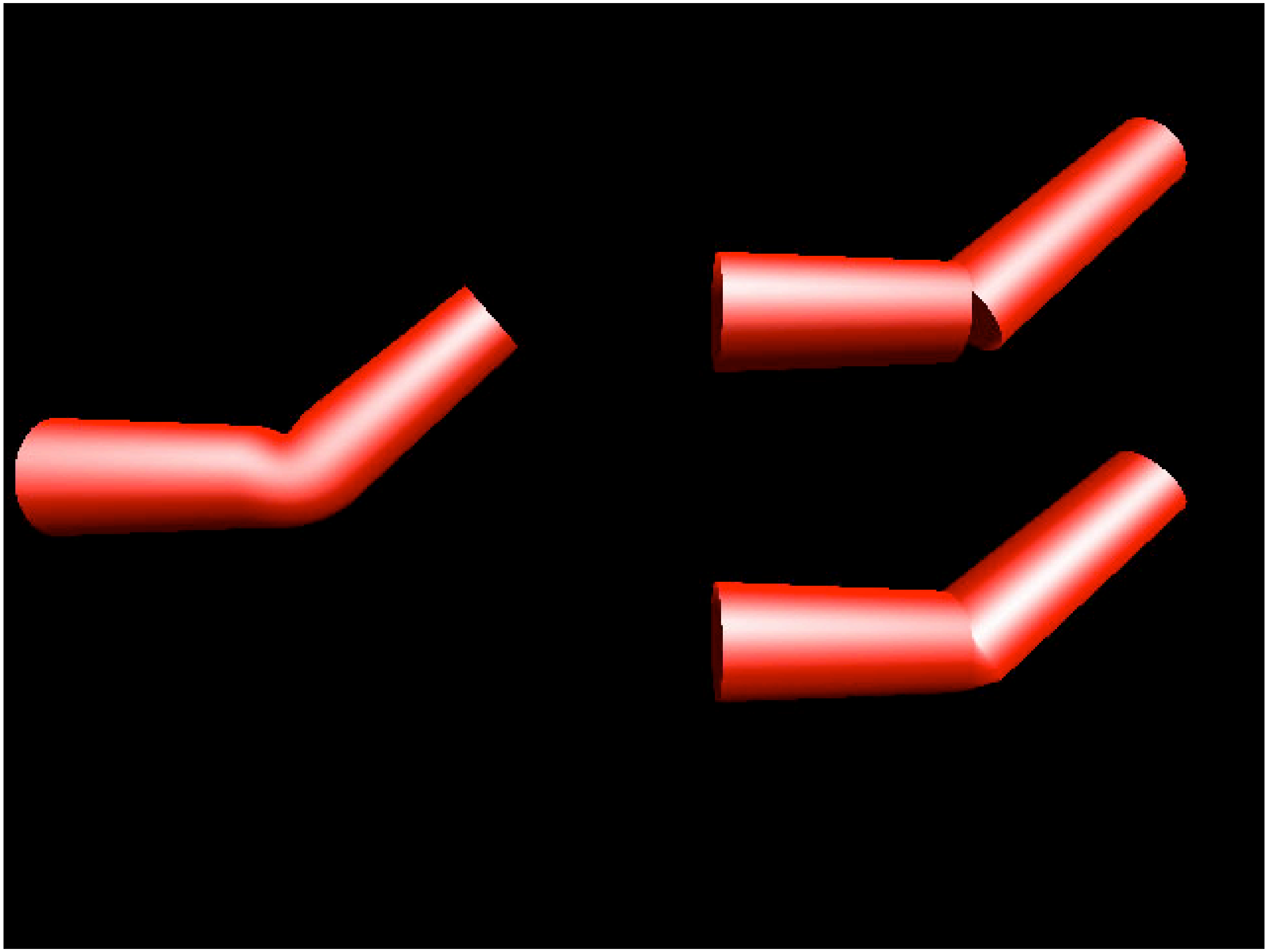
Skinning

- arm example:
  - M1: matrix for upper arm
  - M2: matrix for lower arm



Skinning

-



Example
by NVIDIA

Skinning

- in general:
  - many different matrices make sense!
    - EA facial animations: up to 70 different matrices (“bones”)
    - hardware supported:
      - number of transformations limited by available registers and max. instruction count of vertex programs
      - but dozens are possible today

Fragment Shader Motivation

- idea of per-fragment shaders not new
  - Renderman is the best example, but not at all real time
- traditional pipeline: only major per-pixel operation is texturing
  - all lighting, etc. done in vertex processing, *before* primitive assembly and rasterization
  - in fact, a fragment is *only* screen position, color, and tex-coords
    - normal vector info is not part of a fragment, nor is world position
- what kind of shading interpolation does this restrict you to?

Fragment Shader Generic Structure

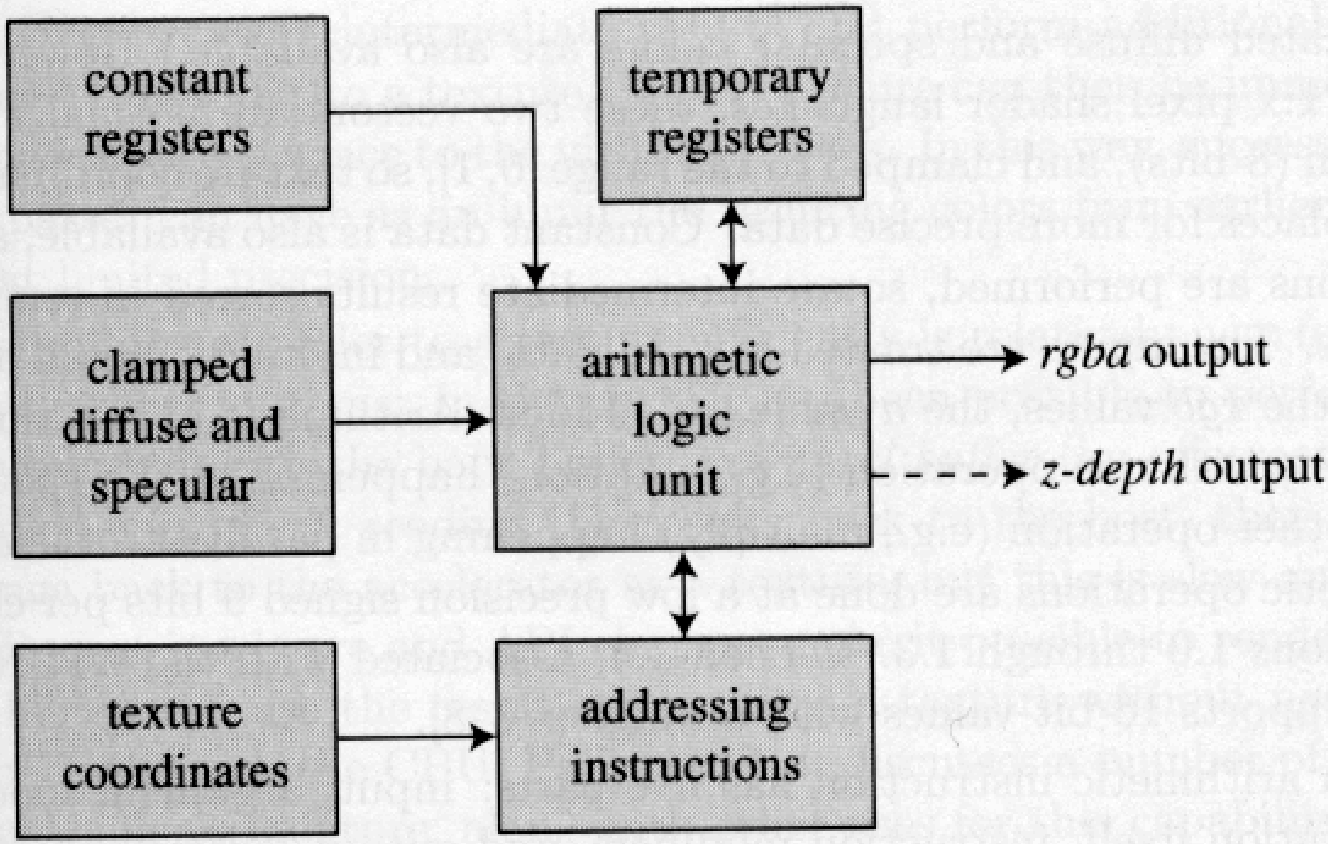


Figure 6.20. Generalized pixel shader. Variants in the pixel shader language primarily affect the way texture address instructions work, where temporary results can be stored, and whether the z-depth can be modified and output.

Fragment Shaders

- fragment shaders operate on fragments in place of texturing hardware
  - after rasterization
  - before any fragment tests or blending
- input: fragment, with screen position, depth, color, and set of texture coordinates
- access to textures, some constant data, registers
- compute RGBA values for fragment, and depth
  - can also kill a fragment (throw it away)
- two types of fragment shaders
  - register combiners (GeForce4)
  - fully programmable (GeForceFX, Radeon 9700)

Fragment Shader Functionality

- consider requirements for Phong shading
  - how do you get normal vector info?
  - how do you get the light?
  - how do you get the specular color?
  - how do you get the world position?

Shading Languages

- programming shading hardware still difficult
  - akin to writing assembly language programs

Vertex Program Example

- *#blend normal and position*
- *# $v = \alpha v_1 + (1-\alpha) v_2 = \alpha(v_1 - v_2) + v_2$*
- **MOV** `R3, v[3] ;`
  MOV `R5, v[2] ;`
  ADD `R8, v[1], -R3 ;`
  ADD `R6, v[0], -R5 ;`
  MAD `R8, v[15].x, R8, R3`
  MAD `R6, v[15].x, R6, R5 ;`
- *# transform normal to eye space*
  DP3 `R9.x, R8, c[12] ;`
  DP3 `R9.y, R8, c[13] ;`
  DP3 `R9.z, R8, c[14] ;`
- *# transform position and output*
  DP4 `o[HPOS].x, R6, c[4] ;`
  DP4 `o[HPOS].y, R6, c[5] ;`
  DP4 `o[HPOS].z, R6, c[6] ;`
  DP4 `o[HPOS].w, R6, c[7] ;`
- *# normalize normal*
  DP3 `R9.w, R9, R9 ;`
  RSQ `R9.w, R9.w ;`
  MUL `R9, R9.w, R9 ;`
- *# apply lighting and output color*
  DP3 `R0.x, R9, c[20] ;`
  DP3 `R0.y, R9, c[22] ;`
  MOV `R0.zw, c[21] ;`
  LIT `R1, R0 ;`
  DP3 `o[COL0], c[21], R1`
  `;`

Vertex Programming Example

- example (from Stephen Cheney)
  - morph between a cube and sphere while doing lighting with a directional light source (gray output)
  - cube position and normal in attributes (input) 0,1
  - sphere position and normal in attributes 2,3
  - blend factor in attribute 15
  - inverse transpose model/view matrix in constants 12-14
    - used to transform normal vectors into eye space
  - composite matrix is in 4-7
    - used to convert from object to homogeneous screen space
  - light dir in 20, half-angle vector in 22, specular power, ambient, diffuse and specular coefficients all in 21

Shading Languages

- programming shading hardware still difficult
  - akin to writing assembly language programs
- shading languages and accompanying compilers allow users to write shaders in high level languages
- examples
  - Microsoft’s HLSL (part of DirectX 9)
  - Nvidia’s Cg (compatable with HLSL)
  - OpenGL Shading Language
  - (Renderman is ultimate example, but not real time)

Cg

- Cg is a high-level language developed by NVIDIA
  - looks like C or C++
  - actually a language and a runtime environment
    - can compile ahead of time, or compile on the fly
  - what it can do is tightly tied to the hardware

Vertex Program Example

```
void C5E2v_fragmentLighting(float4 position  : POSITION,
                            float3 normal    : NORMAL,

                        out float4 oPosition : POSITION,
                        out float3 objectPos : TEXCOORD0,
                        out float3 oNormal   : TEXCOORD1,

                    uniform float4x4 modelViewProj)
{
  oPosition = mul(modelViewProj, position);
  objectPos = position.xyz;
  oNormal = normal;
}
```

Pixel Program Example

```
void C5E3f_basicLight(float4 position    : TEXCOORD0,
                      float3 normal      : TEXCOORD1,

                  out float4 color       : COLOR,

              uniform float3 globalAmbient,
              uniform float3 lightColor,
              uniform float3 lightPosition,
              uniform float3 eyePosition,
              uniform float3 Ke,
              uniform float3 Ka,
              uniform float3 Kd,
              uniform float3 Ks,
              uniform float  shininess)
{
  float3 P = position.xyz;
  float3 N = normalize(normal);

  // Compute the emissive term
  float3 emissive = Ke;

  // Compute the ambient term
  float3 ambient = Ka * globalAmbient;

  // Compute the diffuse term
  float3 L = normalize(lightPosition - P);
  float diffuseLight = max(dot(N, L), 0);
  float3 diffuse = Kd * lightColor * diffuseLight;

  // Compute the specular term
  float3 V = normalize(eyePosition - P);
  float3 H = normalize(L + V);
  float specularLight = pow(max(dot(N, H), 0),
                            shininess);
  if (diffuseLight <= 0) specularLight = 0;
  float3 specular = Ks * lightColor * specularLight;

  color.xyz = emissive + ambient + diffuse + specular;
  color.w = 1;
}
```

Cg Runtime

- sequence of commands to get your Cg program onto the hardware

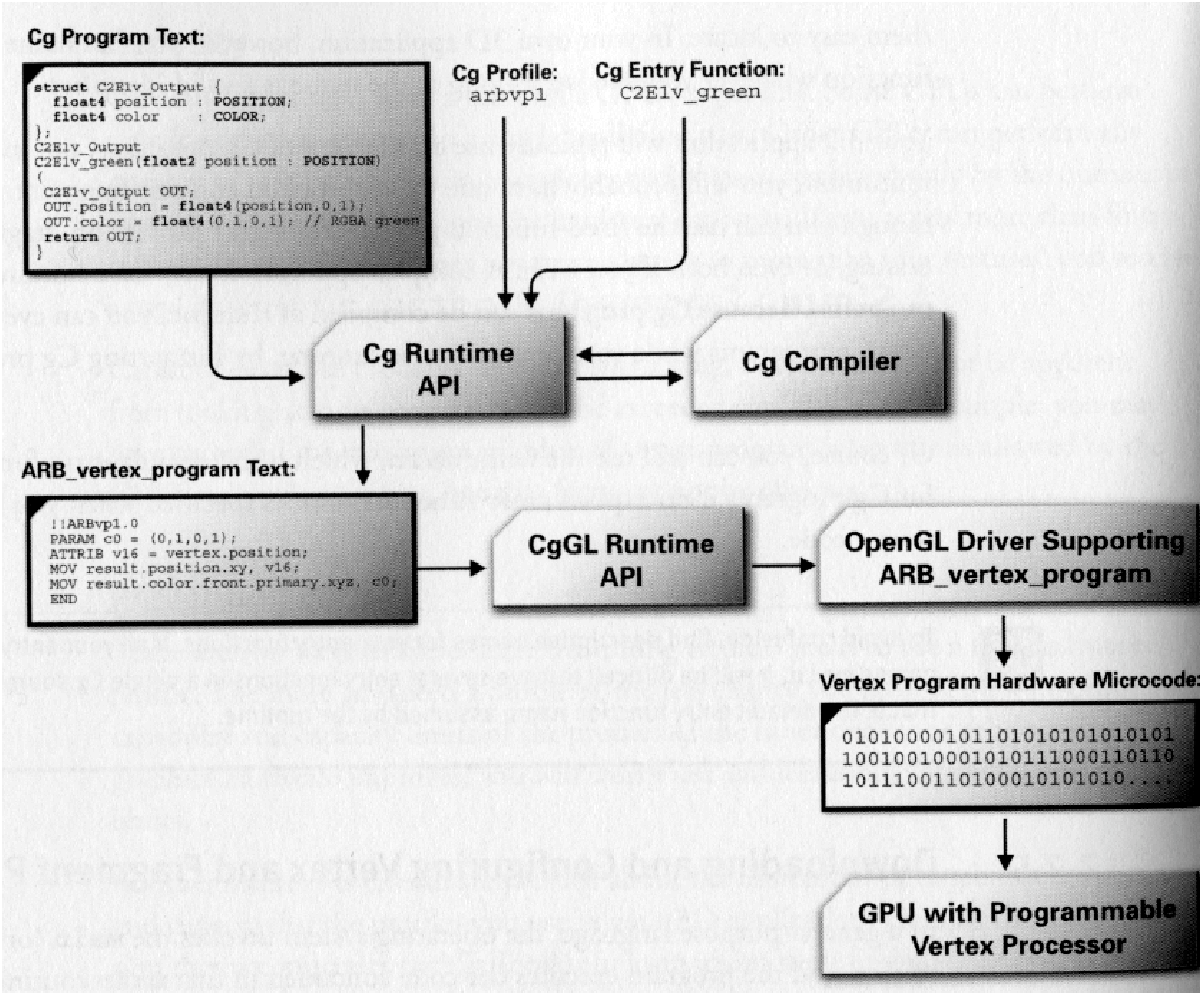


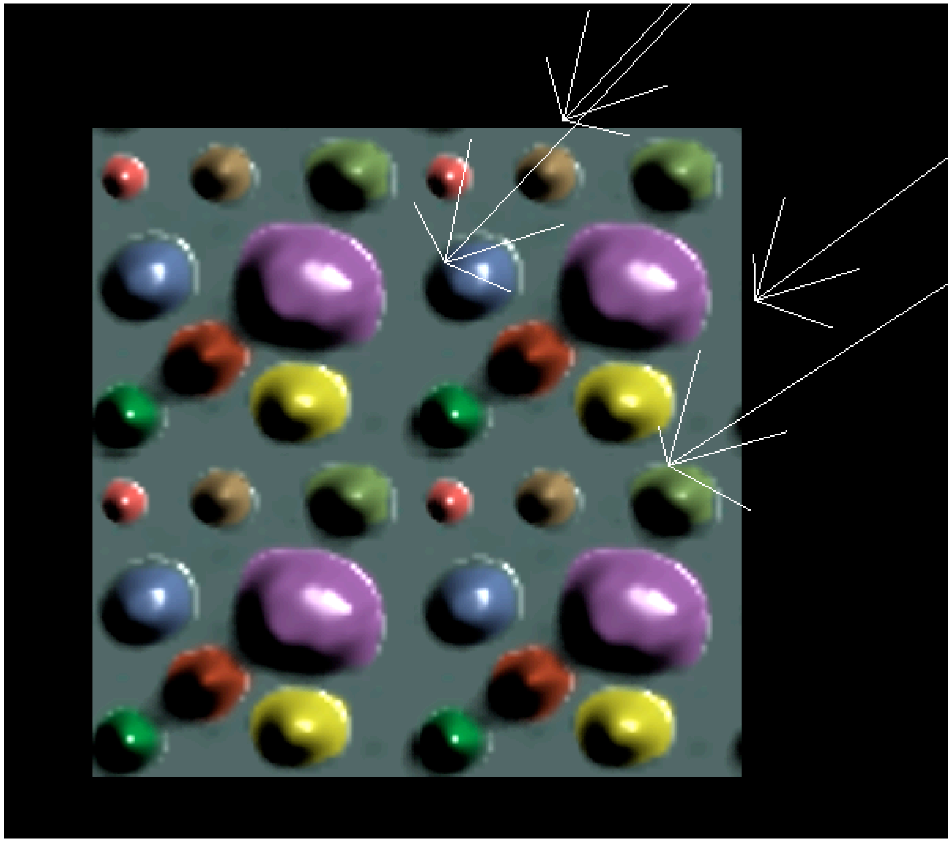
Figure 2-2. Compiling and Loading a Cg Program into the GPU

Bump Mapping

- normal mapping approach:
  - directly encode the normal into the texture map
    - (R,G,B)= (x,y,z), appropriately scaled
  - then only need to perform illumination computation
    - interpolate world-space light and viewing direction from the vertices of the primitive
      - can be computed for every vertex in a vertex shader
      - get interpolated automatically for each pixel
    - in the fragment shader:
      - transform normal into world coordinates
      - evaluate the lighting model

Bump Mapping

- examples



GPGPU Programming

- General Purpose GPU
  - use graphics card as SIMD parallel processor
  - textures as arrays
  - computation: render large quadrilateral
  - multiple rendering passes

Image Formats

- major issue: lossless vs. lossy compression
  - JPEG is lossy compression
  - do not use for textures
    - loss carefully designed to be hard to notice with standard image use
    - texturing will expose these artifacts horribly!
    - can convert to other lossless formats, but information was permanently lost

Acknowledgements

- Wolfgang Heidrich
  - http://www.ugrad.cs.ubc.ca/~cs314/WHmay2006/