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

Review: Ray Casting Traversal Schemes
- Common
  - **pipeline**
    - clipping
    - frame buffer
    - rendering
    - lighting
    - texture mapping
    - shading

Review: Transfer Functions To Classify
- ray casting
  - image order, forward viewing
  - splatting
    - object order, backward viewing
- texture mapping
  - object order
  - back-to-front compositing

Review: Volume Rendering Algorithms
- ray casting
  - image order, forward viewing
  - splatting
    - object order, backward viewing
  - texture mapping
  - object order
  - back-to-front compositing

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 labels
- 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

Review: Ray Casting Traversal Schemes
- *Intensity Max Average Accumulate First*
  - Depth

Review: Premultiplying Colors
- specify opacity with alpha channel: (r,g,b,α)
  - α=1: opaque, α=.5: translucent, α=0: transparent
- A ⊗ B
  - C = C + (1−α)B
  - but what if B is also partially transparent?
  - C = C + (1−α)B + (1−α′)αB
  - γ = β + (1−α′)α
  - 3 multiplies, different equations for alpha vs. RGB
- premultiplying by alpha
  - C' = C, B' = B, A' = αA
  - C' = B' + A'B'
  - γ = β + α
  - 1 multiply to find C, same equations for alpha and RGB

Review: Blending/Compositing
- how might you combine multiple elements?
  - foreground color A, background color B
  - A ⊗ B
  - C = C + (1−α)B
  - but what if B is also partially transparent?
  - C = C + (1−α)B + (1−α′)αB
  - γ = β + (1−α′)α
  - 3 multiplies, different equations for alpha vs. RGB
Modern GPU Features

Skinning

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

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

- example:

  Transition zone:
  - weight for M1 between 0..1
  - weight for M2 between 0..1

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)

- 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

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 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 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)

- 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...

- arithmetic operations on 4-vectors:
  - ADD, MUL, MAD, MIN, MAX, DP3, DP4
  - operations on scalars
  - RCP (1/x), RSQ (1/(x^2)), 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

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

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

- 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

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

- 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...

- arithmetic operations on 4-vectors:
  - ADD, MUL, MAD, MIN, MAX, DP3, DP4
  - operations on scalars
  - RCP (1/x), RSQ (1/(x^2)), 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

- 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

- 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

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

**Fragment Shaders**
- Fragment shaders operate on fragments in place of textures (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)
    - Full 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
- Traditional pipeline: only major per-pixel operation is texturing
- All lighting, etc. done in vertex processing, before primitive assembly and rasterization

**Vertex Program Example**

```
#blend normal and position
DP4 R9.x, R3.v[0], -R5;
DP4 R8, v[15].x, R8, R3;
DP4 R6, v[0], -R5;
R3, v[3];
```

**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)
    - Sphere position and normal in attributes
  - 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 (compatible 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

**Cg Runtime**
- Sequence of commands to get your Cg program onto the hardware

**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

**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

**GPGPU Programming**
- General purpose GPU
  - Use graphics card as SIMD parallel processor
  - Textures as arrays
  - Computation: render large quadrilateral
  - Multiple rendering passes

**Pixel Program Example**

```
void GLUv3N.fragmentLighting(float4 position : POSITION, float3 normal : NORMAL, out float4 oPosition : POSITION, out float3 oNormal : NORMAL, out float3 oObjectPos : TEXCOORD0, out float3 oNormal : TEXCOORD1, uniform float4x4 modelViewProj) {
  // Position + Mask
  oPosition = mul(modelViewProj, position);
  // Position + Normal
  oNormal = normal;
  // Position + Object Position
  oObjectPos = position;
  // Normal + Object Position
  oNormal = mul(modelViewProj, normal);
  // Position + Normal + Object Position
  oPosition = mul(modelViewProj, position);
}
```
Acknowledgements

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