Picking, Recent GPU Developments, Shadows

**CPSC 314**

---

The Rendering Pipeline

- Geometry Database
- Model/View Transform.
- Lighting
- Perspective Transform.
- Clipping

---

**Edge Equation Scan Conversion**

\[(x', y') = \left( \frac{a_1 x + b_1}{a_3 x + b_3}, \frac{a_2 y + b_2}{a_3 x + b_3} \right)\]

---

**Point Sampling**

*Multiply sample grid by image intensity to obtain a discrete set of points, or samples.*

---

**Blending**

*How might you combine multiple elements?*
- New color \( A \), old color \( B \)

---
Premultiplying Colors

Specify opacity with alpha channel \( \langle r,g,b,a \rangle \)
- \( a=1 \): opaque, \( a=0 \): translucent, \( a=0 \): transparent

\[ A \times \overline{B} \]
- \( C = cA \times (1-a)B \)

But what if \( B \) is also partially transparent?
- \( C = cA \times (1-\overline{B}) \)
- \( C = cA \times (1-\overline{B}) + cA \times (\overline{B} \cdot a) \)
- \( \gamma = \beta + (1-\alpha) = \beta + a - \alpha \)
- 3 multiplies, different equations for alpha vs. RGB

Premultiplying by alpha
- \( C = \gamma C, B' = \overline{B} A' = cA \)
- \( C = B' + A' \cdot B' \)
- \( \gamma \times \beta + \alpha - \gamma \)
- 1 multiply to find \( C \), same equations for alpha and RGB

OpenGL Blending

**In OpenGL:**
- Enable blending
  - `glEnable( GL_BLEND )`
- Specify alpha channel for colors
  - `glColor4f( r, g, b, alpha )`
- Specify blending function
  - E.g.: `glBlendFunc( GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA )`
  - \( C = \alpha \text{new} \cdot \text{Cnew} + (1-\alpha \cdot \text{Cold} \)

Double Buffering

**Framebuffer:**
- Piece of memory where the final image is written
- Problem:
  - The display needs to read the contents, cyclically, while the GPU is already working on the next frame
  - Could result in display of partially rendered images on screen
- Solution:
  - Have TWO buffers
    - One is currently displayed (front buffer)
    - One is rendered into for the next frame (back buffer)

**Front/back buffer:**
- Each buffer has both color channels and a depth channel
  - Important for advanced rendering algorithms
  - Doubles memory requirements!

**Switching buffers:**
- At end of rendering one frame, simply exchange the pointers to the front and back buffer
- GLUT toolkit: `glutSwapBuffers()` function
  - Different functions under windows/X11 if not using GLUT

Picking: Interactive Object Selection

**Move cursor over object, click**
- How to decide what is below?

**Ambiguity**
- Many 3D world objects map to same 2D point

**Common approaches**
- Manual ray intersection
- Bounding extents
- Selection region with hit list (OpenGL support)

Manual Ray Intersection

**Do all computation at application level**
- Map selection point to a ray
- Intersect ray with all objects in scene.

**Advantages**
- No library dependence
**Manual Ray Intersection**

*Do all computation at application level*
- Map selection point to a ray
- Intersect ray with all objects in scene.

**Advantages**
- No library dependence

**Disadvantages**
- Difficult to program
- Slow: work to do depends on total number and complexity of objects in scene

---

**Bounding Extents**

**Disadvantages**
- Low precision
- Must keep track of object-rectangle relationship

**Extensions**
- Do more sophisticated bound bookkeeping
  - First level: box check, second level: object check

---

**OpenGL Picking**

*“Render” image in picking mode*
- Pixels are never written to framebuffer
- Only store IDs of objects that would have been drawn

**Procedure**
- Set unique ID for each pickable object
- Call the regular sequence of glBegin/glVertex/glEnd commands
  - If possible, skip glColor, glNormal, glTexCoord etc. for performance

---

**Select/Hit**

**OpenGL support**
- Use small region around cursor for viewport
- Assign per-object integer keys (names)
- Redraw in special mode
- Store hit list of objects in region
- Examine hit list

---

**Viewport**

**Small rectangle around cursor**
- Change coord sys so fills viewport

**Why rectangle instead of point?**
- People aren’t great at positioning mouse
  - Fitts’s Law: time to acquire a target is function of the distance to and size of the target
- Allow several pixels of slop
**Viewport**

**Tricky to compute**
- Invert viewport matrix, set up new orthogonal projection

**Simple utility command**
- `gluPickMatrix(x,y,w,h,viewport)`
  - `x,y`: cursor point
  - `w,h`: sensitivity/slop (in pixels)
- Push old setup first, so can pop it later

**Name Stack**

- "names" are just integers
  `glInitNames()`
- flat list
  `glLoadName(name)`
- or hierarchy supported by stack
  `glPushName(name), glPopName`
  - can have multiple names per object

**Render Modes**

`glRenderMode(mode)`

- `GL_RENDER`: normal color buffer
  - `default`
- `GL_SELECT`: selection mode for picking
- `(GL_FEEDBACK`: report objects drawn)

**Hierarchical Names Example**

```c
for(int i = 0; i < 2; i++) {
  for(int j = 0; j < 2; j++) {
    glPushMatrix();
    glLoadIdentity();
    glPushName(10.0, 0.0, 10.0);
    glPushMatrix();
    glCallList(vec2); // body
    glCallList(vec2); // head
    glCallList(vec2); // eye
    glPopMatrix();
    glPopMatrix();
  }
}
```

- `http://www.lighthouse3d.com/opengl/picking/`

**Hit List**

- `glSelectBuffer(buffer, *buffer)`
  - `where to store hit list data`
- on hit, copy entire contents of name stack to output buffer.
- hit record
  - `number of names on stack`
  - `minimum and minimum depth of object vertices`
  - depth lies in the z-buffer range [0,1]
  - multiplied by 2^32 - 1 then rounded to nearest int

**Integrated vs. Separate Pick Function**

- **Integrate**: use same function to draw and pick
  - Simpler to code
  - Name stack commands ignored in render mode
- **Separate**: customize functions for each
  - Potentially more efficient
  - Can avoid drawing unpickable objects
Select/Hit

**Advantages**
- Faster
  - OpenGL support means hardware accel
- Only do clipping work, no shading or rasterization
- Flexible precision
  - Size of region controllable
- Flexible architecture
  - Custom code possible, e.g. guaranteed frame rate

**Disadvantages**
- More complex

---

Rendering Pipeline

**So far:**
- Have discussed rendering pipeline as a specific set of stages with fixed functionality

**Modern graphics hardware is 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 Motivation

**Hardware “transform&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

---

Modern GPU Features

**CPSC 314**

---

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 Shaders

**Programmability required for more complicated effects**
- The 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
  - I.e. 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 the resulting color

IO for Vertex Shaders (Circa 2001)

- Newer hardware has more instructions, more memory

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

**Concept:**
- 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

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 doll products
  - Enough memory for 24 matrices
  - Can arbitrarily deform objects
    - Procedural freeform deformations
  - Lots of other applications
    - Shading
    - Refraction
    - …
**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

**Skinning**

*Example was one case of general problem:*

- 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**

*Arm Example:*

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

---

**Vertex Program Example**

```
#blend normal and position
if vec4(x, y, z, v) = c(x, y, z, v)
    MDV x, y, z, v[2];
    MDV x, v[3];
    ADD x, v[2], -v[2];
    ADD x, v[3], -v[3];
    MUL x, v[15], x, R5, R2
    MUL x, v[12], x, R6, R2
    # transform normal to eye space
    DPF x, y, c[22];
    DPF y, y, c[12];
    DPF x, x, c[21];
    # transform position and output
    DPF x[0], R9, c[1];
    DPF x[1], R9, c[1];
    DPF x[2], R9, c[1];
    # normalize normal
    DPF x, w, R9, R9;
    R3D R9, w, R9, w;
    MUL R9, w, R9, w;
    # apply lighting and output color
    DPF R9, y, R9, c[22];
    MUL R9, y, R9, c[22];
    LIT R1, R0, c[22];
    DPF x[1], R9, c[22];
```

---

**Skinning**

*Approach:*

- Multiple transformation matrices
  - There is 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 both matrices
- Yields 2 different transformed vertex positions!
- Use per-vertex blending weights to interpolate between the two positions
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

**The idea of per-fragment shaders have been around for a long time**
- Renderrman is the best example, but not at all real time

**In a traditional pipeline, the only major per-pixel operation is texture mapping**
- All lighting, etc. is done in the vertex processing, before primitive assembly and rasterization
- In fact, a fragment is only screen position, color, and tex-coords
  - I may have misled you earlier – 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 Shader Generic Structure](image)

Fragment Shaders

- Fragment shaders operate on fragments in place of the texturing hardware
  - After rasterization, before any fragment tests or blending
- Input: The fragment, with screen position, depth, color, and a set of texture coordinates
- Access to textures and some constant data and registers
- Compute RGBA values for the fragment, and depth
  - Can also "kill" a fragment, that is throw it away
- Two types of fragment shaders: register combiners (GeForce4) and fully programmable (GeForceFX, Radeon 9700)

Fragment Shader Functionality

**At a minimum, we want to be able to do Phong interpolation**
- 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 is still a difficult process**
- Akin to writing assembly language programs

**Shading languages and accompanying compilers allow users to write shaders in high level languages**
- Two examples: Microsoft’s HLSL (part of DirectX 9) and Nvidia’s Cg (compatible with HLSL)
  - Renderman is the ultimate example, but it’s not real time
Cg

Cg is a high-level language developed by NVIDIA

- It looks like C or C++
- Actually a language and a runtime environment
  - Can compile ahead of time, or compile on the fly
  - Why compile on the fly?
- What it can do is tightly tied to the hardware
  - How does it know which hardware, and how to use it?

Vertex Program Example

```c
void CSERV_FragmentLighting(float4 position : POSITION,
                           float3 normal : NORMAL,
                           out float4 cPosition : POSITION,
                           out float3 cObjectPos : POSITION,
                           out float3 cNormal : POSITION,
                           uniform float4x4 modelViewProj)
{
  cPosition = mul(modelViewProj, position);
  cObjectPos = position.
  cNormal = normal;
}
```

Pixel Program Example

```c
void CSERV_TextureFloat(float2 TexCoord : TEXCOORD) 
{
  // Do something with TexCoord...
}
```

Cg Runtime

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

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

**Examples**

- Vertex Program Example
- Cg Runtime
- Bump Mapping
  - Normal Mapping Approach
  - Examples

© Wolfgang Heidrich
Shadow Maps

Approach for shadows from point light sources
- Surface point is in shadow if it is not visible from the light source
- Use depth buffer to test visibility:
  - Render scene from the point light source
  - Store resulting depth buffer as texture map
  - For every fragment generated while rendering from the camera position, project the fragment into the depth texture taken from the camera, and check if it passes the depth test.

Shadow Maps

Shadow maps using the alpha test

Alpha shadow map:

Shadow map applied as projective texture

Light source 2
- Computed in fragment shader
**Shadow Maps**

*Subtract shadow map from computed depth*
- Also in fragment shader
- Wherever the result is not 0, we have shadow!

**Shadow Volumes**

*Use new buffer: stencil buffer*
- Just another channel of the framebuffer
- Can count how often a pixel is drawn

**Algorithm (1):**
- Generate silhouette polygons for all objects
  - Polygons starting at silhouette edges of object
  - Extending away from light source towards infinity
  - These can be computed in vertex programs

**Algorithm (2):**
- Render all original geometry into the depth buffer
  - I.e. do not draw any colors (or only draw ambient illumination term)
- Render front-facing silhouette polygons while incrementing the stencil buffer for every rendered fragment
- Render back-facing silhouette polygons while decrementing the stencil buffer for every rendered fragment
- Draw illuminated geometry where the stencil buffer is 0, shadow elsewhere

**Shadow Maps**

*Discussion*
- Same precision problems as with depth buffer
  - Objects can wrongly cast shadows onto themselves due to numerical problems ("surface acne")
- Finite resolution
  - Shadows look blocky (from linear texture interpolation!) at large distance from the light source
**Shadow Volumes**

*Discussion:*
- Object space method therefore no precision issues
- Lots of large polygons: can be very slow
  - High geometry count
  - Large number of pixels rendered

**Coming Up...**

*Wednesday:*
- Color

*Friday:*
- Ray-tracing