Johns Hopkins
Engineering for Professionals

605.767 Applied Computer Graphics

Brian Russin

JOHNS HOPKINS
WHITING SCHOOL
of ENGINEERING

Module 11C
Pipeline Optimizations

Application Optimization

- Application OptimizationTry different compiler optimization flags
  - Example: a) Minimize Code Size vs. b) Optimize for Speed
    - Sometimes get better performance with a) due to caching performance
- Human code optimization may also be necessary
  - Use code profiler to find “hot spots”
    - Places where code spends the most time
    - e.g., Intel VTune (performance analyzer) or Quantify
  - Re-examine algorithm and code syntax
    - Try different variants
- Most graphics applications are CPU limited

Code Optimization

- Use Single Instruction Multiple Data (SIMD) instruction sets
  - 2-4 elements can be processed in parallel
    - Excellent for vector operations
- Avoid using division
  - Can take 4 – 39 times as long as other instructions
  - Vector::Normalize: Find 1/d then multiply each component
- Unroll small loops
  - Avoids loop overhead
  - Caution: code size is larger and may degrade caching performance
- Align frequently used data structures to 32 byte boundaries on PC
  - Pad data structures
  - Linux: cacheprof can help identify caching bottlenecks
- Use approximations to expensive math methods like cos, sin, tan, exp, pow, sqrt
  - If lower precision is acceptable

Code Optimization (cont.)

- Inline small, frequently used methods
- Lessen floating point precision
  - Remember to use f at the end of constants (x = 3.14f)
    - Otherwise whole expressions may cast to double
- Use lower precision when possible
  - To lessen data size sent down the pipeline
- Use const where possible
  - Helps compiler with optimization
- Pass structs by reference rather than value
- Try different methods of coding the same algorithm
  - Pre-increment (++n) rather than post-increment (n++) is usually faster, unless a copy is needed
    - Same for pre-/post-decrement
  - Test indexing vs. pointer increments
    - p[n] = q[n] vs. *p++ = *q++

Memory Issues

- Often there are memory caching issues
  - Things like pointer indirection and function calls can sometimes cause cache misses that stall the CPU
  - Exploit the cache(s) of the architecture
    - Cache prefetching (can be difficult)
- Try different organizations of data structures
  - Array of structures vs. structure of arrays
    - Different ones work better on different architectures
  - Memory accessed sequentially in code should be stored sequentially in memory
- System memory allocation and deallocation can be slow on some systems
  - Often better to allocate a big pool of memory at start and use your own allocation / free methods to get memory from the preallocated
  - Avoid malloc/free within rendering loop
    - Allocate scratch space once, have arrays that only grow

Optimizing the Graphics API Calls

- Somewhat dependent on GPU driver
- Every draw call has some overhead associated with it
    - Practical limit of number of GPU draw calls per frame
- Proper organizing the API calls
    - Organize scene graph to minimize expensive calls
    - Most-expensive to least-expensive state changes (abbreviated list)
        - Render Target (framebuffer): ~60K/s
        - Program (shaders): ~300K/s
        - Texture Binding: ~1.5M/s
            - Consider grouping textures into arrays or packing a single buffer to reduce switching
        - Uniform Buffer Bindings
        - Vertex Bindings
        - Uniform Buffer Updates: ~10M/s
    - http://media.steampowered.com/apps/steamdevdays/slides/beyondporting.pdf

Geometry Optimization

- Use connected primitives where possible
  - Triangle strips and fans are more efficient than individual triangles
- Indexed vertex arrays are most efficient
  - Indexed primitives often use pre and post Transform and Lighting caches
    - Eliminates multiple transformation and lighting of shared vertices
- New methods becoming available for providing compressed forms for vertex data
  - E.g., 3 bytes for vertex normals (rather than 3 floats)
    - Vertex shaders allow decompression in geometry stage
- Eliminate small batches
  - Larger vertex arrays are sometimes more efficient than several groups of smaller arrays

Lighting Optimization

- Lighting can use most of the geometry processing
- Efficiency considerations include:
  - Is lighting needed for all polygons?
    - Disable lights or all lighting when not needed
    - Often don't need to light large, background polygons
    - Turn off if decal texture is being used
  - Directional lights are faster than point lights are faster than spotlights
  - More light sources -> less speed
  - Two-sided lighting is more expensive than one-sided lighting
  - Non-local viewer is more efficient
    - When used with directional lights, H is constant over entire scene
    - Can cause slight shifts in location of specular highlights
  - Lighting requires normals to be transformed and renormalized
    - If object is not scaled the normals can be precalculated

$$I = E_m + I_a k_a + \sum_j f_{attj} S_j \left[I_{aj} k_a + I_{dj} k_d (L \cdot N) + I_{sj} k_s (H \cdot N)^n \right]$$

Prelit Objects

- If light sources are static and material has no specular component the ambient and diffuse components can be precomputed
  - Attach colors to the vertices
    - More data is required for object and more data sent over the bus
  - Can turn off lighting for these objects
  - Often called **preshaded**, **prelit**, or **baked**
- Can also use more elaborate methods like radiosity to precompute lighting
  - Stored as colors at the vertices or as lightmaps
- More difficult to add specular components since they are view dependent

Optimizing the Rasterization Stage

- Enable backface culling for closed solid objects or objects that never show their back faces
  - Adds cost to determine if polygon is front facing
    - If most polygons are front facing can actually slow down the geometry stage
- Disable depth buffer when not needed
  - After framebuffer has been cleared any background image does not require depth testing
- Textures may be too big: can overload texture memory and cause cache misses
  - Use texture sizes no bigger than necessary
- Avoid expensive internal texture formats
- Try less expensive texture filtering
- For line drawing, turn off smooth shading if not being used

Depth Complexity

- Depth complexity is the number of times a pixel is touched during rendering
  - Often instructional, tells a lot about load on the rasterization stage
  - High depth complexity may benefit from occlusion culling techniques
- https://web.tecgraf.puc-rio.br/~ismael/Cursos/Cidade_CG/labs/OpenGL/OpenGL_siggraph1998/node244.html
  - Describes how to generate an image with depth complexity by using the stencil buffer

  1. Clear the depth and stencil buffer; glClear(GL_STENCIL_BUFFER_BIT|GL_DEPTH_BUFFER_BIT).
  2. Enable stenciling; glEnable(GL_STENCIL_TEST).
  3. Set up the proper stencil parameters; glStencilFunc(GL_ALWAYS, 0, 0), glStencilOp(GL_KEEP, GL_INCR, GL_INCR).
  4. Draw the scene.
  5. Read back the stencil buffer with glReadPixels(), using GL_STENCIL_INDEX as the format argument.
  6. Draw the stencil buffer to the screen using glDrawPixels() with GL_COLOR_INDEX as the format argument.

Overall Optimization

- Reduce the total number of primitives passing through entire pipeline
  - Simplify models (appropriate level of detail)
  - Culling techniques (view-frustum, occlusion, etc.)
- Preprocess models
  - E.g., tessellate concave and non-simple polygons in advance
- Choose as low a precision as possible for vertices, normals, colors, texture coordinates
  - Lower precision means less memory, data moves quicker through the pipeline
    - Often can choose between short, int, float, double
- Turn off features not in use
  - Depth buffering, fog, blending, texturing
    - E.g., depth buffering in first primitive drawn after depth buffer clear
- Use fast path if available
  - Some architectures have a highly optimized path
    - E.g., primitive sizes that are optimized for the architecture

Overall Optimization (cont.)

- Make sure textures reside in texture memory (if possible) to avoid swapping
- Minimize state changes
  - Group objects with similar rendering state together
    - Texture, material, lighting, transparency, etc.
  - State changes sometimes cause a flush of the graphics pipeline
- Call glGet during setup and avoid during scene drawing
- Separate 2D from 3D operations
  - May be expensive to switch between them
- Avoid frame buffer reads (expensive)
- Avoid multiple passes by using multi-texturing, if possible