Overview

- Locating the bottleneck
- Performance measurements
- Optimizations
- Balancing the pipeline
- Other optimizations: multi-processing, parallel processing

Pipeline Optimization

"We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil"

– Donald Knuth

- Make it run first, then optimize
- But only optimize where it makes any difference
- Pipeline Optimization: Process to maximize the rendering speed, then allow stages that are not bottlenecks to consume as much time as the bottleneck.

Pipeline Optimization

- Stages execute in parallel
- Always the slowest stage is the bottleneck of the pipeline
- The bottleneck determines throughput (i.e., maximum speed)
- The bottleneck is the average bottleneck over a frame
- Cannot measure intra-frame bottlenecks easily
- Bottlenecks can change over a frame
- Most important: find bottleneck, then optimize that stage!

Locating the Bottleneck

- Two bottleneck location techniques:
- Technique 1:
  - Make a certain stage work less
  - If performance is the better, then that stage is the bottleneck
- Technique 2:
  - Make the other two stages work less or (better) not at all
  - If performance is the same, then the stages not included above is the bottleneck
- Complication: the bus between CPU and graphics card may be bottleneck (not a typical stage)

Application (CPU) Stage the Bottleneck?

- Use top, osview command on Unix, TaskManager on Windows.
- If app uses (near) 100% of CPU time, then very likely application is the bottleneck
- Using a code profiler is safer.
- Make CPU do less work (e.g., turn off collision-detection)
- Replace glVertex and glNormal with glColor
- Makes the geometry and rasterizer do almost nothing
- No vertices to transform, no normals to compute lighting for, no triangles to rasterize
- If performance does not change, program is CPU-bound, or CPU-limited

Geometry Stage the Bottleneck?

- Trickiest stage to test
- Why? Change in geometry workload usually changes application and rasterizer workload.
- Number of light sources only affects geometry stage:
  - Disable light sources (vertex shaders can make this simple).
  - If performance goes up, then geometry is bottleneck, and program transform-limited
- Alternately, enable all light sources; if performance stays the same, geometry stage NOT the bottleneck
- Alternately, test CPU and rasterizer instead

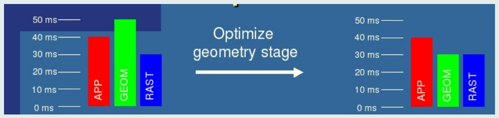
Rasterizer Stage the Bottleneck?

- The easiest, and fastest to test
- Simply, decrease the size of the window you render to
  - Does not change app. or geometry workload
  - But rasterizer needs to fill fewer pixels
  - If the performance goes up, then program is “fill-limited” or “fill-bound”
- Make rasterizer work less: Turn of texturing, fog, blending, depth buffering etc (if your architecture have performance penalties for these)

Optimization

- Optimize the bottleneck stage
- Only put enough effort, so that the bottleneck stage moves
- Did you get enough performance?
  - Yes! Quit optimizing
  - NO! Continute optimizing the (possibly new) bottleneck
- If close to maximum speed of system, might need to turn to acceleration techniques (spatial data structures, occlusion culling, etc)

Illustrating Optimization



- Height of bar: time it takes for that stage for one frame
- Highest bar is bottleneck
- After optimization: bottleneck has moved to APP
- No use in optimizing GEOM, turn to optimizing APP instead

Application Stage Optimization

- Initial Steps:
  - Turn on optimiziation flags in compiler
  - Use code profilers, shows places where majority of time is spent
  - This is time consuming stuff
- Strategy 1: Efficient code
  - Use fewer instructions
  - Use more efficient instructions
  - Recode algorithmically
- Strategy 2: Efficient memory access

Appliction:Code Optimization Tricks

- SIMD intstructions sets perfect for vector ops
  - 2-4 operations in parallell
  - SSE, SSE2, 3DNow! are examples
- Division is an expensive operation
  - Between 4-39 times slower than most other instructions
  - Good usage Example: vector normalization:
    Instead of

    $$v = (v_x/d, v_y/d, v_z/d)$$

    Do

    $$d = \mathbf{v} \cdot \mathbf{v}, f = 1/d, \mathbf{v} = \mathbf{v} * f$$

- On some CPUs there are low-precision versions of $(1/x)$ and square root reciprocal $(1/\sqrt{x})$

Code Optimization Tricks (contd)

- Conditional branches are generally expensive;
  - Avoid if-then-else if possible
  - Sometimes branch prediction on CPUs works remarkably well
- Math functions (sin, cos, tan, sqrt, exp, etc.) are expensive
  - Rough approximation might be sufficient
  - Can use first few terms in Taylor series
- Inline code is good (avoids function calls)
- float (32 bits) is faster than double (64 bits); less data is sent down the pipeline

Code Optimization Tricks (contd)

- Compiler optimization: Hard to predict: –counter vs. counter–
- Use const in C and C++ to help to compiler with optimization
- Following often incur overhead:
  - Dynamic casting (C++)
  - Virtual methods
  - Inherited constructors
  - Passing structs by value

Memory Optimization

- Memory hierarchies (caches) in modern computers - primary, secondary caches.
- Bad memory access pattern can ruin performance
- Not really about using less memory, though that can help

Memory Optimization Tricks

- Sequential access: Store data in order in memory:
  - Tex Coords #0, Position #0, Tex Coords #1, Position #1, Tex coords #2, Position #2, etc.
- Cache prefetching is good, but hard to control
- malloc() and free() may be slow: Consider using a custom storage allocator - allocate memory to a pool at startup

Memory Optimization Tricks (contd)

- Align data with size of cache line
  - Example: on most Pentiums, the cache line size if 32 bytes
  - Now, assume that it takes 30 bytes to store a vertex
  - Padding with another 2 bytes to 32 bytes will likely perform better.
- Following pointers (linked list) is expensive (if memory is allocated arbitrarily)
  - Does not use coherence well that cache usually exploits
  - That is, the address after the one we just used is likely to be used soon
  - Paper by Smits on ray tracing shows this.

Geometry Stage: Optimization

- Geometry stage does per-vertex ops
  - Best way to optimize: Use Triangle strips!!!
- Lighting optimization:
  - Spot lights expensive, point light cheaper, directional light cheapest
  - Disable lighting if possible
  - Use as few light sources as possible
  - If you use $1/d^2$ fallof, then if $d > 10$ (example), disable light

Geometry Stage: Optimization

- Normals must be normalized to get correct lighting
  - Normalize them as a preprocess, and disable normalizing if possible
- Lighting can be computed for both sides of a triangle; disable if not needed.
- If light sources are static with respect to geometry, and material is only diffuse
  - Precompute lighting on CPU
  - Send only precomputed colors (not normals)

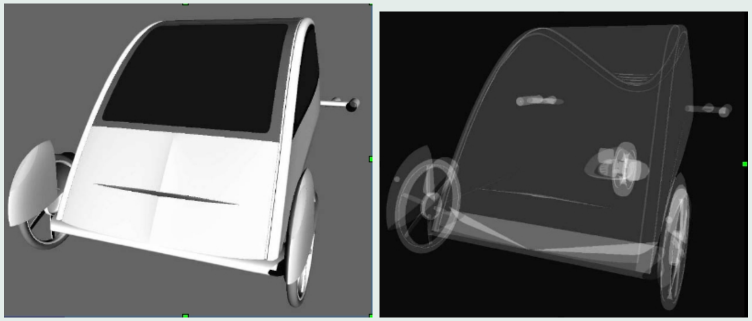
Raster Stage: Optimization

- Rasterizer stage does per-pixel ops
- Simple Optimization: turn on backface culling if possible
- Turn off Z-buffering if possible:
  - Example: after screen clear, draw large background polygon
  - Using polygon-aligned BSP trees
- Draw in front-to-back order
- Try disable features: texture filtering mode, fog, blending, multisampling

Raster Stage: Optimization

- To make rasterization faster, need to rasterize fewer (or cheaper) pixels:
  - Make window smaller
  - Render to a smaller texture, and then enlarge texture onto screen
- Depth complexity is number of times a pixel has been written to
  - Good for understanding behaviour of application

Depth Complexity



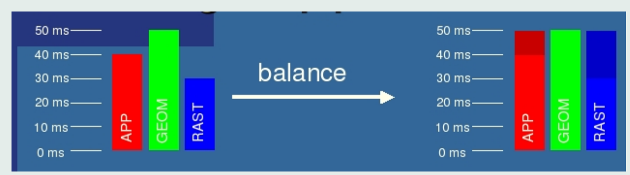
Overall Optimization: General Techniques

- Reduce number of primitives, eg. using polygon simplification algorithms
- Preprocess geometry and data for the particular architecture
- Turn off features not in use such as:
  - Depth buffering, Blending, Fog, Texturing

Overall Optimization (contd)

- Minimize state changes by grouping objects
  - Example: objects with the same texture should be rendered together
- If all pixels are always drawn, avoid color buffer clear
- Frame buffer reads are expensive
- Display lists may work faster
- Precompile a list of primitives for faster rendering
- OpenGL API supports this

Balancing the Pipeline

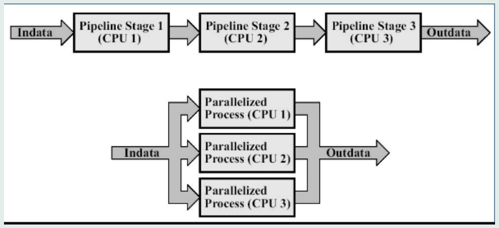


- The bottleneck stage sets the frame rate
- The other two stages will be idle for some time
- Also, to sync with monitor, there might be idle time for all stages
- Exploit this time to make quality of images better if possible

Balancing the Pipeline

- Increase number of triangles (affects all stages)
- More lights, more expensive (geometry)
- More realistic animation, more accurate collision detection (application)
- More expensive texture filtering, blending, etc. (rasterizer)
- If not fill-limited, increase window size
- Note: there are FIFOs between stages (and at many other places too) to smooth out idleness of stages
- More techniques in text.

Multiprocessing



- Use this if application is bottleneck, and is affordable
- Two major ways: (1) Multiprocessor pipelining, (2) Parallel processing

Summary

- Pipeline optimization is no substitute for good algorithms!
- Do optimization as a last step.
- Primarily for products that should be shipped
- Most often good to use triangle strips!