When Light Speed Isn't Fast Enough

Advanced Optimization Techniques

Fabian 'ryg' Giesen

farbrausch / .theprodukkt

Assembly '05

What this seminar is about

- You want to solve a certain problem efficiently.
- You have already exhausted algorithmic optimizations.
- But “light speed” isn’t fast enough!
- There are techniques that can still help you:
  - Architecture-specific optimizations.
  - Changing the problem you’re trying to solve.
  - Maybe you don’t need everything you think you do!
- This is what this seminar is about.

Prerequisites

This is an "advanced" seminar, so I assume...

- You have already written programs yourself.
- You have tried (successfully?) to optimize them.
- You have certain basic knowledge:
  - Elementary data structures (linked lists, trees)
  - Fundamental algorithms
- You can read C++-like pseudocode.

Memory Optimizations

- Memory is often a limiting factor to performance.
- So you should try to use your memory efficiently.
- The most important thing is using the *Cache* properly.

Cache recap

- Small amount of very fast “short-term” memory
- Actually multiple cache levels, I’ll idealize that away :)
- Organized in so-called *Lines* of typically 64 bytes.
- On a memory access, the processor. . .
  - *Somehow* calculates the cache address for that memory.
  - Checks whether the correct line is in the cache.
  - Loads it (and throws another one out) if it’s not.
  - Performs the memory access inside the cache.
- The CPU always loads/stores *whole* cache lines.
- Cache *misses* (a cache line needs to be fetched) are slow.

Cache design principles: General

So, if you want to keep the cache happy:

- Keep your access patterns nice and predictable!
  - *Sequential* memory accesses are best.
  - Lots of pointer chasing is bad for the cache.
- Think carefully on size/alignment of data structures.
  - Multiples/Divisors of cache line size are best.
- Try to concentrate writes on small regions of memory
  - As said, CPU always writes *whole* cache lines!
  - So don't just change one byte here, two bytes there
  - Don't write at all if you don't have to.
- Don't touch data you don't really need.

Cache design principles: Hot and cold data

- For any given algorithm, there are two types of data:
  - *Hot data* is data that is used frequently.
  - *Cold data* is used seldomly or not at all.
- Hot data is what actually benefits from caching.
- So make sure your hot data is cached efficiently:
  - Don't mix hot and cold data in the same part of memory.
  - Try to make sure hot data doesn't leave the cache.
  - Reorganize your data structures if necessary.

Time for an example...

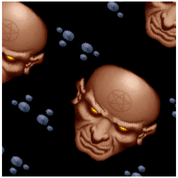
Okay, that's enough theory for now, it's time for a proper example...

Fast rot-zoomers

- ???
- Back to the 90s?
- Yeah, it's "out of fashion" now, but...
  - It's easy to see what's going on.
  - It will show us some nice techniques.
  - Generalizations of those techniques are very relevant today.

The general algorithm

$\Longrightarrow$



- You have your current *u* and *v* coordinate.
- Go through pixels of the screen in turn:
  - Get pixel at position (u, v) from source image.
  - Store that pixel to the screen.
  - Every x step, add *dudx* to *u* and *dvdx* to *v*.
  - Every y step, add *dudy* to *u* and *dvdy* to *v*.
- There's no obvious way to make this any simpler.

Cases that work well

- No rotation, no zoom:

 $\Longrightarrow$ 

  - Just sequential reads, so no problems.

- No rotation, zoom in ($\times 2$):

 $\Longrightarrow$ 

  - Even better, everything gets reused.

Problem cases

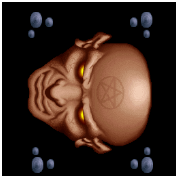
- No rotation, zoom out ($\times 0.5$):

$\Longrightarrow$

  - Only half of the pixels read get used!

- 90° rotation, no zoom:

$\Longrightarrow$



  - Every pixel fetched is from a different line in the image.
  - $\Rightarrow$ Cache miss almost every time

Fixing the problems: Zoom out

- As we saw, when zooming out, cache usage suffers.
- Well, just make sure you don't zoom out then!
- Use smaller textures when zooming out $\Rightarrow$ *Mipmaps*
- This is the actual reason we use mipmapping for realtime 3D.
  - Improved image quality is a nice bonus though :)
- Idea also applies to other problems:
  - Instead of skipping through your data in big steps, use a coarser representation.

Fixing the problems: Rotation

- Idea by Niklas Beisert (pascal/Cubic Team), 1995:
- Don't render the image line by line, but in 8×8 pixel blocks!
  - Main problem was that we went too far in just one direction.
- 3D cards reorder the texture instead ("*Swizzling*")
  - Same idea, same effect (better cache usage).
  - Somewhat harder to visualize/code though.
- All that said, there's not much difference on current CPUs.
  - Caches have gotten a lot bigger since 1995.
  - Cache logic is better, too.
  - You need unrealistic texture sizes to see a difference.
- But the techniques discussed are still relevant:
  - Storing your data in multiple resolutions.
  - Cache-friendly traversal matters.

AGP Writes

- Suppose you're coding some morphing/pulsating object.
- Whatever 3D API you prefer, you'll use a dynamic *Vertex Buffer (Object)* if you care for performance.
- That vertex buffer will usually lie in AGP memory.
- The object will change every frame.

AGP Writes (2)

Code

```
for(int i=0;i<nVerts;i++) {
  Vector3 pos,normal;
  Vector2 uv;

  // Calculate pos,normal,uv with magic jizzblob formula.

  buffer[i].pos = pos;
  buffer[i].normal = normal;
  buffer[i].uv = uv;
}
```

- Say you don't change UV coords, and want to skip writing them.
- Bad idea!

AGP Writes (3)

- Remember what I said about cache lines?
  - "They are loaded from memory on the first access."
  - Oops, I lied there :)
- Different parts of memory get cached differently.
- AGP Memory is usually tagged as *Write Combined*.
  - Reads are not cached.
  - But adjacent writes are combined so whole cache lines get written when possible.
- This only works if you don't leave holes!
- So don't skip bytes when writing to AGP memory.
- Combined writes are *a lot* faster than partial ones!

Simplified Memory Management

- In, say, a 3D engine, you have a lot of dynamic data per frame:
  - List of visible objects
  - List of active lights
  - Instanced animation data
  - etc.
- Most of this comes in relatively small structures.
  - So you get thousands of small allocations/frame.
- Standard allocators (`new`) ain't very good at that.
  - Relatively slow.
  - Not memory efficient for small objects.
  - Things get spread over a big region of memory.
- Can't we do better than that?

Simplified Memory Management (2)

- Turns out we can.
- Just allocate, say, 200k, then manage them yourself.
- How to manage it efficiently?
- As simple as possible!
- Just use it as a stack.
- Bare-bones version fits onto one slide:

Simplified Memory Management (3)

MemStack class

```
class MemStack {
  char *Buf,*Current;

public:
  MemStack(int size) { Buf = Current = new char[size]; }
  ~MemStack()        { delete[] Buf; }
  void Reset()       { Current = Buf; }

  void *Alloc(int amount) {
    void *ptr = Current;
    Current += amount;
    return ptr;
  }
};
```

- In practice, you'll add a typesafe Alloc() template.

What's good about this

- Memory management doesn't get any shorter than that.
- It doesn't get any faster or simpler, either.
- No bookkeeping overhead worth mentioning.
- You don't have to `delete` - just `Reset()` every frame.
- Things allocated after each other get memory next to each other.
- $\Rightarrow$ Good cache behavior.

Problems

- Limited to objects that have a lifetime of one frame.
  - Or however often you call `Reset()`.
- Don't try this when you have constructors or destructors.
- Static maximum amount of memory you can use.
  - This can be fixed rather easily, though.
- Only usable for small data structures.
  - Unless you're willing to make the buffer unreasonably large.

Conclusion

- Not a technique usable everywhere.
- But so simple it's always worth trying.
- When it works, it shines.

Sorting tricks

- Don't worry, I won't start a boring lecture now.
- This is about different forms of sorting than you usually learn about in CS classes.
  - Not comparision-based.
  - Linear running time (not $O(n \log n)$).
  - Don't always give a correct solution.
- What the heck is "incorrect" sorting good for?
  - You get correct sorting up to a certain tolerance.
    - No huge differences, just "a little bit off".
  - Sometimes, there's no easy way to determine a "correct" sorting anyway (e.g. Z-Sorting triangles)
  - Sometimes, it's not critical things are perfectly sorted (e.g. sorting by Material in a 3D Engine)

Ordering Tables

- Name originates (to my knowledge) from the PS1.
- A very interesting piece of hardware:
  - Hardware rasterization (not perspective correct).
  - Simple fixed-point vector unit (GTE).
  - 33MHz MIPS R3000 CPU.
  - No Z-Buffer!
- You had to sort the triangles yourself.
  - Facesorting on a 33MHz CPU?
- Luckily, the PS1 engineers came up with a nice solution: *Ordering Tables*.

Ordering Tables (2)

- The idea is very simple:
  - Divide the Z Range into N equal-size parts.
  - Create an array of N pointers to polygons.
- Every frame, clear that array to zero.
- For every triangle you want to draw:
  - Calculate the corresponding Z-Index i.
  - Insert it into $Order[i]$ like with a linked list.
- Finally, go through the array in reverse (back to front), painting triangles as you traverse.
  - The PS1 could do this in hardware.

Ordering Tables (3)

Example code

```
Triangle *Order[256];

void Clear() { memset(Order,0,sizeof(Order)); }

void Insert(Triangle *tri) {
  int index = (tri->averageZ-zNear)*256/(zFar-zNear);
  tri->next = Order[index];
  Order[index] = tri;
}

void Paint() {
  for(int i=255;i>=0;i--)
    for(Triangle *tri=Order[i];tri;tri=tri->next)
      DrawTriangle(tri);
}
```

Ordering Tables: Wrap-up

- Again, very nice when you can get away with it.
- Perfect for "approximate front-to-back sorting" of objects.
  - Don't use it to sort translucent polygons though.
  - *All* types of face sorting are impractical on current GPUs.
  - Use *Render Passes* instead ($\rightarrow$ Chaos' seminar)
- Application of a general strategy called "Bucket Sort".

Bucket Sort

- Throw your data into “buckets” based on some ordering.
- Sort elements of each bucket individually.
  - Or just don’t.
- Traverse all buckets in order to get sorted list.
- Only works when you can calculate bucket index quickly.
  - No problem for numeric data, strings.
- Repeated application leads to a “proper” sorting algorithm.
  - $\Rightarrow$ Straight Radix Sort.
  - Maybe in another seminar, enough algorithms for 45 minutes :)

That's it!

Questions?

ryg (at) theprodukkt (dot) com
http://www.farbrausch.de/~fg