SSE

Motivation

- We'll look at raytracing with SIMD
- This will show us some additional ways SSE is applied to solve larger problems

Option

- We could just use SSE operations for our vector math
- Ex: v1 + v2
  - Load v1 $\rightarrow$ XMM register
  - Load v2 $\rightarrow$ XMM register
  - Add
- This is not ideal. Why not?

Problem

- XMM register can hold 4 floats, but we're adding two vec3's
- This wastes 25% of our system's capacity: We ignore the last slot

Problem

- Operations like dot product are more difficult
  - Requires adding "horizontally" in single XMM register
  - SSE/AVX is generally not so good at horizontal operations

SIMD

- Option: *Data parallel computing*
  - We will process four pixels simultaneously
  - Same instructions for each pixel but different data (ray directions)
  - This is where SIMD has its best chance to shine
  - This is how GPU compute shaders work too!

SIMD

- Need to process four adjacent pixels at a time
- First, define xmm and xmmi types to make code easier to work with
- Files: xmm.h, ymm.h

Code

- Review: Non-SIMD raytracing code:

```
//s=ray start, v=ray direction. We return closest intersection point (ip) & normal at intersection (N)
bool traceTriangles(const vector<Triangle>& triangles, const vec3& s, const vec3& v, vec3& ip, vec3& N){
  float closestT = 1E99;
  int closestIndex = -1, idx = -1;
  for(auto& T : triangles ){
        idx++;
        float denom = dot(T.N,v);  //if denom is zero, we get t=infinity
        float numer = -(T.D + dot(T.N,s) );
        float t = numer/denom;
        if( t < 0 )            continue;
        if( t >= closestT )    continue;
        vec3 vv = t*v;
        vec3 v0 = T.p[0]-s, v1 = T.p[1]-s, v2 = T.p[2]-s;
        if( scalarTripleIsNegative( vv, v0,v2))
            continue;
        if( scalarTripleIsNegative( vv, v1,v0))
            continue;
        if( scalarTripleIsNegative( vv, v2,v1))
            continue;
        closestIndex = idx;
        closestT = t;
        ip = s + vv;
  }
  N = triangles[closestIndex].N;
  return (closestIndex != -1);
}
```

Note

- We have a helper function:

```
bool scalarTripleIsNegative(const vec3& a, const vec3& b, const
    vec3& c){
     return dot(cross(a,b),c)<0.0f;
}
```

Strategy

- We'll trace four [SSE] or eight [AVX] pixels at once
- So we need to change the function so it takes four rays as input and returns four pixels' worth of output

```
bool traceTriangles(const vector<Triangle>& triangles, const vec3&
    s, const array<vec3,4>& v, array<vec3,4>& ip, array<vec3,4>& ,
    array<vec3,4>& color){
```

Changes

- We need to alter the closestT variable so we keep track of four t values:
  xmm closestTs(1E99);
- Then we have the loop. It's just like before:

```
for(auto& T : triangles ){
    idx++;
    ...
}
```

Loop

- In the loop, we have our first statement:
  float denom = dot(T.N,v); //if denom is zero, we get t=infinity
- We need to do the four dot products in parallel
- What would this look like?

Dot

- Recall:

```
float dot(vec3& v, vec3& w){
    return v.x*w.x + v.y*w.y + v.z*w.z;
}
```

Dot

- float denom = dot(T.N,v);
- Idea: Load T.N to three xmm's: One will have {T.N.x, T.N.x, T.N.x, T.N.x}, one will have {T.N.y, T.N.y, T.N.y, T.N.y}, and the last will have T.N.z, replicated four times
- Load the four v.x's to an xmm
  - Repeat with y's and z's
- Then do the math

Dot

- Loading everything:

```
xmm Nx(T.N.x);
xmm Ny(T.N.y);
xmm Nz(T.N.z);
xmm vx(v[0].x,v[1].x,v[2].x,v[3].x);
xmm vy(v[0].y,v[1].y,v[2].y,v[3].y);
xmm vz(v[0].z,v[1].z,v[2].z,v[3].z);
```

Multiply

- We can now compute the four dot products
- Again, old code:
  `float denom = dot(T.N,v);`
- New code:
  `xmm denoms = Nx*vx + Ny*vy + Nz*vz;`

Numerator

- To compute the numerators:
  `float numer = -(T.D + dot(T.N,s) );`
- All the terms here are the same for all four rays
- So we can compute as:
  `xmm numers = xmm( -(T.D + dot(T.N,s) ) );`

Quotient

- This is really easy:
  xmm ts = numers/denoms;

Next

- We then have two tests: Old code:

```
if( t < 0 )
    continue;
```

- But: We seem to have hit a snag
  - We have four rays we're processing
  - Can't just do a single check! Some rays might say "true" and some might say "false"

Pattern

- We use the *masking* pattern
- Compute a mask:

```
xmm mask = (ts >= xmm::allzeros());
```

  - Notice that we've reversed the sense of the test!
- Slot i of mask is 0 if ray i failed the test
- Slot i of mask is all 1's if ray i passed the test
- We continue on without looking at mask...

Test 2

- We have a second test: Old code:

```
if( t >= closestT )
    continue;
```

- We *accumulate* this into our mask (again, reversing the test):

```
mask = mask & (ts < closestT);
```

Tests

- We have more tests. But first, compute some preliminary values
- Old code:

```
vec3 vv = t*v;
vec3 v0 = T.p[0]-s, v1 = T.p[1]-s, v2 = T.p[2]-s;
```

- New code:

```
xmm vvx = ts*vx, vvy=ts*vy, vvz=ts*vz;
vec3 v0 = T.p[0]-s, v1 = T.p[1]-s, v2 = T.p[2]-s;
```

- Notice: v0, v1, v2 are the same for all four rays
  - So we can use the same code as before

Tests

- More tests!
- Original code:

```
if( scalarTripleIsNegative( vv, v0,v2))
    continue;
```

- Again, we accumulate this into our mask (reversing the test):

```
mask = mask & scalarTripleIsNotNegative( vvx, vvy, vvz, v0, v2 );
```

  - Do we need to go over how to rewrite scalarTripleIsNegative?

Tests

- You can probable guess where this is going...
- Change

```
if( scalarTripleIsNegative( vv, v1,v0))
    continue;
```

- To:

```
mask = mask & scalarTripleIsNotNegative( vvx, vvy, vvz, v1,v0 ));
```

Tests

- And the last test:

```
if( scalarTripleIsNegative( vv, v2,v1))
    continue;
```

- Change as before...

Mask

- Finally, we're ready to use the mask
- Our old code:

```
ip = s + vv;
closestT = t;
closestIndex = idx;
```

- But: This code was never executed if we hit any of the "continue" statements earlier in the loop
- So we use the mask to say "only change the value if we still have nonzero in a given slot"

Note

- We need to declare variables outside the loop:

```
xmm closestipx, closestipy, closestipz;
xmmi closestIndices(-1);
```

Closest Index

- Update closest indices conditionally:

```
closestIndices = blend( closestIndices, xmmi(idx), mask );
```

- Same for closestTs

IP

- Intersection point: Old code:

```
ip = s + vv;
```

- New code: We must compute the x, y, and z components of the intersection points separately

```
closestipx = blend( closestipx, xmm(sx)+vvx );
...similar for y and z...
```

Finally

- Conditionally blend:

```
closestipx = blend( closestipx, ipx, mask );
closestipy = blend( closestipy, ipy, mask );
closestipz = blend( closestipz, ipz, mask );
```

Explanation

- If mask retained a ’true’ value for slot i, then closestip[i] and closestIndices[i] get updated
- Else, they retain their old values
  - Remember: For blend(a,b,v): If high bit of v[i] is zero: Choose a[i] else choose b[i]

Color

- After we're out of the for-loop over the triangles, we need to return the color values so we can compute the colors
  - We could use SSE to do color computation, but since there's relatively few operations (exactly one shading per pixel), it shouldn't be a big problem to use non-SSE code
  - But if we wanted to squeeze every drop of performance out of the hardware, we would do this in SIMD too
- We just need to recombine the separate x,y,z values

Recombination

```
float ipxf[4], ipyf[4], ipzf[4];
int ci[4];
closestipx.store(ipxf);
closestipy.store(ipyf);
closestipz.store(ipzf);
closestIndices.store(ci);
for(int i=0;i<4;++i){
    ip[i]=vec3(ipxf[i], ipyf[i], ipzf[i] );
    N[i] = triangles[ci[i]].N;
}
```

Results

- Results: -O3, Core i7-5500U, 2.4GHz
  - Non-SSE: 8.44 seconds
  - SIMD, SSE (four at a time): 4.47 seconds (speedup = 1.89x)
  - SIMD, AVX (eight at a time): 2.67 seconds (speedup = 3.16x)

Analysis

- Why is SSE not 4x faster?
  - Some overhead from setting up vector operations
  - Some “Non-SSE” code actually does use SSE (compiler auto-vectorizes when it knows how to do it)
  - Not doing all operations with SSE/AVX (ex: Color computations)

Assignment

- Implement parallel raytracing of triangle meshes using AVX

or

- Implement parallel raytracing of spheres using AVX

or

- Do both for 200%

Sources

- Ray Tracing: Rendering a Triangle (Barycentric Coordinates). https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-rendering-a-triangle/barycentric-coordinates

Created using LaTeX.

Main font: Gentium Book Basic, by Victor Gaultney. See http://software.sil.org/gentium/
Monospace font: Source Code Pro, by Paul D. Hunt. See https://fonts.google.com/specimen/Source+Code+Pro and http://sourceforge.net/adobe
Icons by Ulisse Perusin, Steven Garrity, Lapo Calamandrei, Ryan Collier, Rodney Dawes, Andreas Nilsson, Tuomas Kuosmanen, Garrett LeSage, and Jakub Steiner. See http://tango-project.org