Rendering Worlds with Two Triangles

with raytracing on the GPU

in 4096 bytes

Iñigo Quilez – iq/rgba

August 22 at NVSCENE 08
A bit of context

- Amazing progression in raw GPU power.
- Shaders 3 and 4 flexible enough for
  - Experimenting with new techniques.
  - Revival of some old-school effects (at a higher quality than ever).
- Unexpected benefits:
  - Easy to set up and very compact code.
  - 4k demo coders have jumped into it.
A bit of context

- The idea: draw two triangles that cover the entire screen area, and invoke a pixel shader that will create an animated or static image.

- Make the complete demo self-contained in no more 4096 bytes (that includes the “engine”, music, shaders, animations, textures and everything).
A bit of context

- How much is a kilobyte?

This is the size of a 4 kbytes production

This is the size of a 64 megabytes demo or video
A bit of context

- Probably not a fair comparison (we cannot blame demo coders for being lazy compared to intro coders).
- The “visual beauty” is not a linear function of the size in kilobytes.

- Speculation: With current technology, the optimal “vibes per kilobyte” (aka result/effort ratio, or “wow factor”) is around 100 kb productions.
Index

- Old-school effects are back
- Rendering with distance fields
Index

• Old-school effects are back
• Rendering with distance fields
Old-school effects are back

- Filling the screen with a shader, and producing an image or animation from it, only works for algorithms and effects that follow this pattern:

```cpp
for( each pixel p )
{
    outputColor = doSomething( p );
}
```

- This doesn’t naturally extend to effects that need to do operations across pixels (gather and scatter operations). Multipass techniques can be used, but
  - it might actually be slower than doing it on the CPU
  - it’s not elegant
  - it’s not very compact for 4k demos
Old-school effects are back

• Julia and Mandelbrot sets (the “hello world” of gfx programming)
Old-school effects are back

- Plane deformations

Oldschool software version
```c
for( int i=0; i<numPixel; i++ )
{
    const uint16 offset = magicLUT[i] + time;
    buffer[i] = texture[ offset & 0xffff ];
}
```

Pixel shader version
```c
void main( void ) //for( each pixel p )
{
    vec2 offset = magicFormula(p, time);
    gl_FragColor = texture2D(texture, offset);
}
```
Old-school effects are back

- Others?
  - Rasterizers!
    - Vertex transformation
    - Triangle rasterization
    - Not perspective corrected
- Metaballs
- Plasmas
- Raytracing
Old-school effects are back

- Whitted raytracing of simple scenes/primitives
  - A classic in demoscene
  - With fake analytic Ambient Occlusion

Chocolux, by Auld, 2007 [1 kbyte demo]

Kinderpainter, by rgba, at BCN 2006 [4k kbytes demo]
Old-school effects are back

Source code of chocolux, by Auld (link with Crinkle-r)
Old-school effects are back

- Pathtracing of simple scenes/primitives

Off the shelf, by Loonies, at Breakpoint 2008 [4k kbytes image]

PhotonRace, by Archee, at Buenzli 2008 [4k kbytes image]
Old-school effects are back

- GPU raytracing beyond spheres and planes (I mean polygons)

Images reproduced with permission of Vrcontext (www.vrcontext.com)
Old-school effects are back

• GPU raytracing beyond spheres and planes (I mean polygons)
  • A very hot research topic today (because raytracing is the future...)
  • Difficult to beat CPU raytracers
    • kd-tree/bih/bvh traversal is quite incoherent
    • They all need a stack (unavailable today on shaders).
  • For massive models, streaming to video memory is needed. That makes it more complex.
  • In any case, demosceners have not been interested on real raytracing so far; even less in the 4k categories.
Old-school effects are back

- Raymarching
  - Kind of raytracing for all those objects that don’t have an analytic intersection function.
Old-school effects are back

• Raymarching -- what?
  • Heightmaps
  • Volume textures
  • Procedural isosurfaces
  • Analytic surfaces
Old-school effects are back

- Raymarching -- how?
  - Constant steps
  - Root finders (bisection, Newton-Raphson...)
  - Distance fields

*Failty, by Loonies, 2006, a 4 kbytes demo*

*Tracie, by TBC, 2007, a 1 kbytes demo*

*Kindernoiser, by rgba, 2007, a 4 kbytes demo*
• Old school effects are back
• Rendering with distance fields
Rendering with distance fields
Rendering with distance fields

• Similarly previous works
  • “Ray tracing deterministic 3-D fractals” published at Siggraph 1989 by D.J. Sandin and others.

• The trick is to be able to compute or estimate (a lower bound of) the distance to the closest surface at any point in space.
• This allows for marching in large steps along the ray.
Distance-aided ray marching
Distance-aided ray marching
Distance-aided ray marching
Distance-aided ray marching
Distance-aided ray marching
Distance-aided ray marching
Distance-aided ray marching
Rendering with distance fields

Pros

- Much faster than constant-size stepping.
- Much easier to control than root finders (bisection, Newton...)
- Room for optimization, like using bigger steps when we are further from the ray origin
  - Error in world coordinates decreases as $1/d$
  - So stepping proportionally to $d$ results in constant screen space error.

Cons

- Slow on the boundaries of the objects (hopefully not that many pixels).
- Can control it by imposing a minimum step size.
Rendering with distance fields

- Slicesix needs 50 million evaluations of the very expensive distance function for a 1280x720 pixel image.
- 60% of the evaluations are for primary rays (av. 17 steps per ray).
- 40% of the evaluations are for lighting and shading.

- Note the very expensive marching on the object edges.
Rendering with distance fields

• We need a distance field:
  • Analytic computation (“Ray tracing deterministic 3-D fractals”)
  • Precomputed (static scene) LUT
    • 3D texture (“Per pixel displacement mapping with distance functions”)
    • Octree / KdTree
  • What if we do it 100% procedurally? (“Slisesix”)
Rendering with distance fields

- Procedural distance fields
  - Don’t define the surface first and then compute the distance field, but directly code a distance field and a surface will emerge.
  - Tweak the distance field directly until you get what you want/can.
- Helpful techniques that can be used:
  - Arbitrary combination and instantiation
  - Infinite repetition
  - Deforming space: twisting, bending, deforming
  - Cheap detail surfaces
  - Blend shapes
Rendering with distance fields :: Combination

- Combination of (instanced) distance fields can be done by taking the min of the distance fields involved.
- Instance transformation can be done by inverse transforming the domain (the input to the distance function).

```cpp
float combinedDistanceField( vec3 p )
{
    float dist1 = distanceField_A( M1inv*p );
    float dist2 = distanceField_A( M2inv*p );
    float dist3 = distanceField_B( M3inv*p );
    return min( dist1, min( dist2, dist3 ) );
}
```
Rendering with distance fields :: Domain repetition

- \( \text{dist} = \text{fourMagicColumns}( p.x, p.y, p.z ); \)
Rendering with distance fields :: Domain repetition

- \( \text{dist} = \text{fourMagicColumns}(\mod(p.x,1), p.y, \mod(p.z,1)) \);
Rendering with distance fields :: Domain distortion

float dist = distanceToColumn(p);
Rendering with distance fields :: Domain distortion

```cpp
float twistedColumn( vec3 p )
{
    vec3 q = rotateY(p, p.y*1.7);
    return distanceToColumn(q);
}
```
Rendering with distance fields :: Domain distortion

```cpp
float rr = dot(p.xy, p.xy);
for(int i = 0; i < 6; i++) {
    vec3 q = rotateY(p, TWOPI*i/6.0);
    distance = min(distance, distanceToTheXAxis(q));
}
```
Rendering with distance fields :: Domain distortion

```cpp
float rr = dot(p.xy,p.xy);
for( int i=0; i<6; i++ )
{
    vec3 q = rotateY( p, TWOPI*i/6.0 );
    q.y += 0.6*rr*exp2(-10.0*rr);
    distance = min( distance, distanceToTheXAxis(q) );
}
```
Rendering with distance fields :: Domain distortion

```cpp
float rr = dot(p.xy,p.xy);
for( int i=0; i<6; i++ )
{
    vec3 q = rotateY( p, TWOPI*i/6.0 + 0.4*rr*noise2f(vec3(4*rr,6.3*i)) );
    q.y += 0.6*rr*exp2(-10.0*rr);
    distance = min( distance, distanceToTheXAxis(q) );
}
```
Rendering with distance fields :: Blending fields

```cpp
float distanceToMonster( vec3 p )
{
    float dist1 = distanceToBall(p);
    float dist2 = distanceToTentacles(p);
    float bfact = smoothstep( length(p), 0, 1 );
    return mix( dist1, dist2, bfact );
}
```
Rendering with distance fields :: Adding details

dist = distanceToColmuns(p);
Rendering with distance fields :: Adding details

\[
\text{dist} = \text{distanceToColmuns(p)} + 0.000001\cdot\text{clamp(fbm(p)}, 0, 1)\;
\]
Rendering with distance fields :: Lighting

- Lighting
  - Normals
  - Bump mapping
  - Soft shadows
  - Ambient Occlusion
Rendering with distance fields :: Lighting

- Normals computed by central differences on the distance field at the shading point (gradient approximation).

- Bump map computed by adding the gradient of a fractal sum of Perlin noise functions to the surface normal.
  - $n = \text{normalize}(\text{grad}(\text{distance}, p) + \text{bump} \times \text{grad}(\text{fbm}, p))$;
  - $\text{bump}$ is small and depend on the material.
  - $\text{grad}(\text{func}, p) = \text{normalize}($
    - $\text{func}(p+\{\text{eps},0,0\}) - \text{func}(p-\{\text{eps},0,0\}),$
    - $\text{func}(p+\{0,\text{eps},0\}) - \text{func}(p-\{0,\text{eps},0\}),$
    - $\text{func}(p+\{0,0,\text{eps}\}) - \text{func}(p-\{0,0,\text{eps}\})$ $)$;
Rendering with distance fields :: Ambient Occlusion

- Fake and fast Ambient Occlusion.
- VERY CHEAP, even cheaper than primary rays! Only 5 distance evaluations instead of casting thousand of rays/evaluations.
Rendering with distance fields :: Ambient Occlusion

- In a regular raytracer, primary rays/AO cost is 1:2000. Here, it’s 3:1 (that’s almost four orders of magnitude speedup!).
- It’s NOT the screen space trick (SSAO), but 3D.
- The basic technique was invented by Alex Evans, aka Statix ("Fast Approximation for Global Illumination on Dynamic Scenes", 2006). Greets to him!
- The idea: let $p$ be the point to shade. Sample the distance field at a few (5) points around $p$ and compare the result to the actual distance to $p$. That gives surface proximity information that can easily be interpreted as an (ambient) occlusion factor.
Rendering with distance fields :: Ambient Occlusion
Rendering with distance fields :: Ambient Occlusion
Rendering with distance fields :: Ambient Occlusion
Rendering with distance fields :: Ambient Occlusion
Rendering with distance fields :: Ambient Occlusion

\[ ao = 1 - k \sum_{i=1}^{5} \frac{1}{2^i} (pink_i - yellow_i) \]

\[ ao = 1 - k \sum_{i=1}^{5} \frac{1}{2^i} (i \cdot \Delta - distfield(p + n \cdot i \cdot \Delta)) \]

- The exponential decay is there so further away surfaces occlude less than near by ones.
Rendering with distance fields :: Ambient Occlusion

- Works in realtime too, provided you can compute distances to surfaces.
Rendering with distance fields :: Soft Shadows

- Fake and fast soft shadows.
- Only 6 distance evaluations used instead of casting hundreds of rays.
- Pure geometry-based, not blurring.
- Recipe: take $n$ points on the line from the surface to the light and evaluate the distance to the closest geometry. Find a magic formula to blend the $n$ distances to obtain a shadow factor.
Rendering with distance fields

On a GeForce 8800 GTX, it renders around 20 times faster than on a dual core CPU. It will very soon be realtime.
Rendering with distance fields

- Related info: