Advanced Procedural Rendering in DirectX 11

Matt Swoboda
Principal Engineer, SCEE R&D PhyreEngine™ Team
Demo Coder, Fairlight

GAME DEVELOPERS CONFERENCE
SAN FRANCISCO, CA
MARCH 5-9, 2012
EXPO DATES: MARCH 7-9
2012


FARLIGHT
ANGELDAWN/DUMP
when dreams come true

Aim

- More dynamic game worlds.

Demoscene?

- I make demos
  - “Like games, crossed with music videos”
- Linear, non-interactive, scripted
- All generated in real-time
  - On consumer-level PC hardware
- Usually effect-driven & dynamic
  - Relatively light on static artist-built data
  - Often heavy on procedural & generative content

DirectX 11?

- DirectX 9 is very old
  - We are all very comfortable with it..
  - .. But does not map well to modern graphics hardware
- DirectX 11 lets you use same hardware smarter
  - Compute shaders
  - Much improved shading language
  - GPU-dispatched draw calls
  - .. And much more

Procedural Mesh Generation

A reasonable result from random formulae

(Hopefully a good result from sensible formulae)

Signed Distance Fields (SDFs)

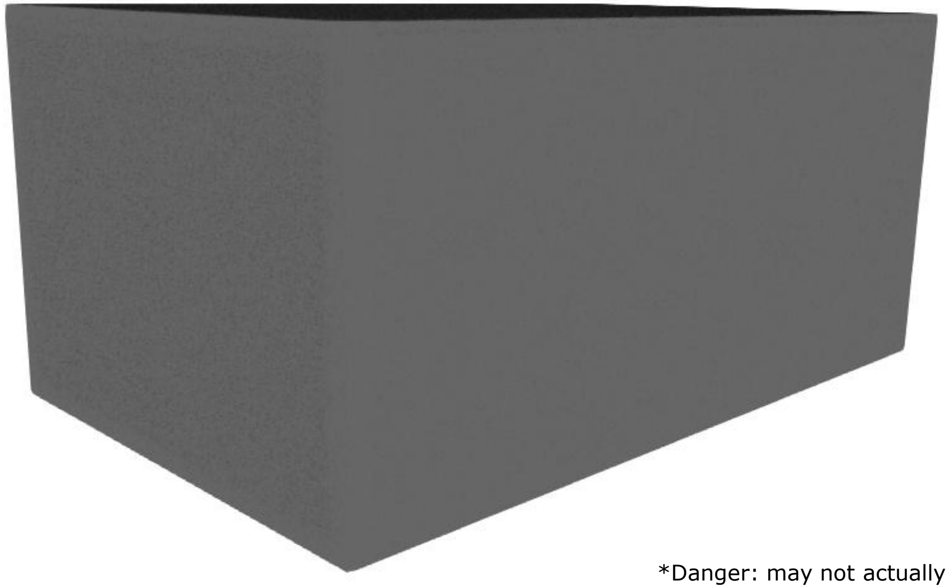
- ***Distance function***:
  - Returns the closest distance to the surface from a given point
- ***Signed distance function***:
  - Returns the closest distance from a point to the surface, positive if the point is outside the shape and negative if inside

Signed Distance Fields

- Useful tool for **procedural geometry creation**
  - Easy to define in code ..
  - .. Reasonable results from “random formulae”
- Can create from meshes, particles, fluids, voxels
- CSG, distortion, repeats, transforms all easy
- No concerns with **geometric topology**
  - Just define the **field** in space, polygonize later

A Box

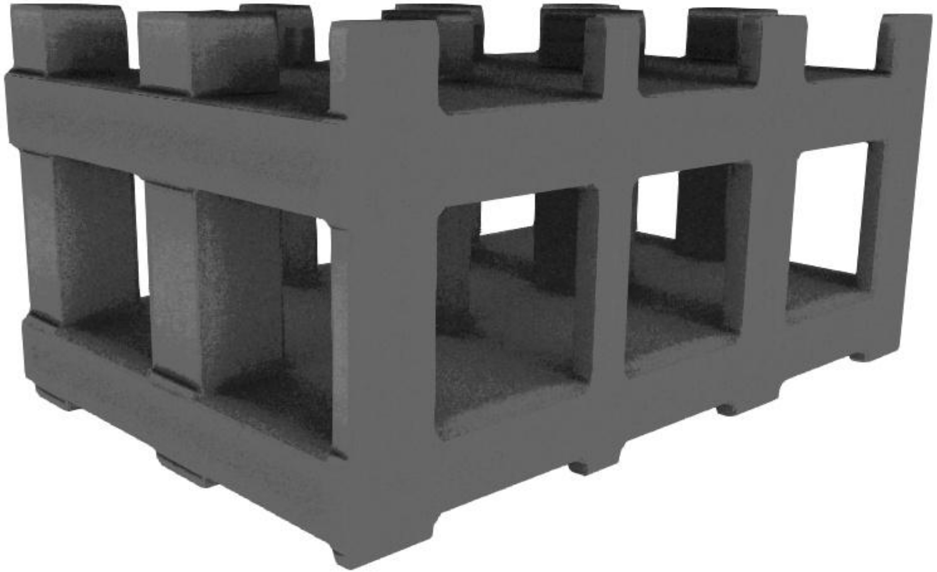
```
Box(pos, size)
{
  a = abs(pos-size) - size;
  return max(a.x,a.y,a.z);
}
```



*Danger: may not actually compile

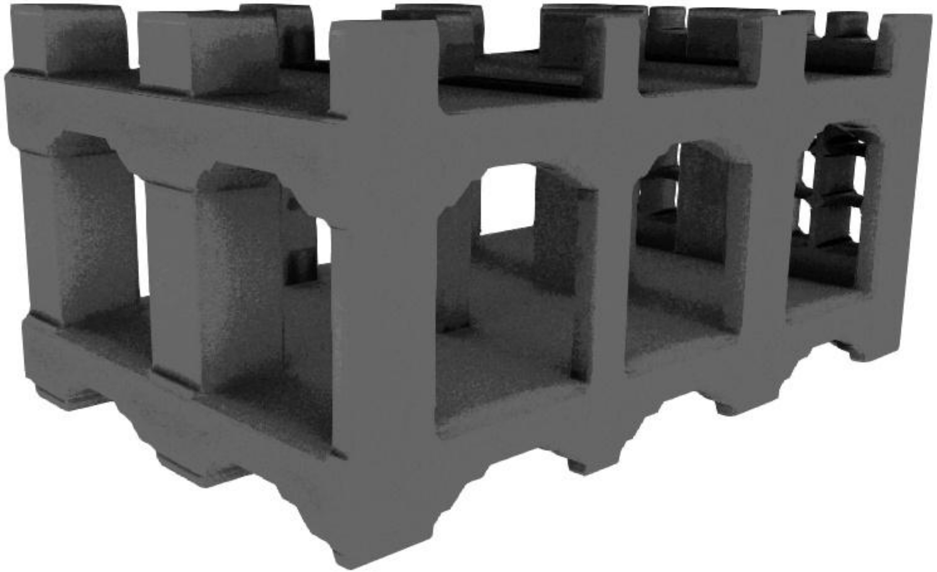
Cutting with Booleans

```
d = Box(pos)
c = fmod(pos * A, B)
subD = max(c.y,min(c.y,c.z))
d = max(d, -subD)
```



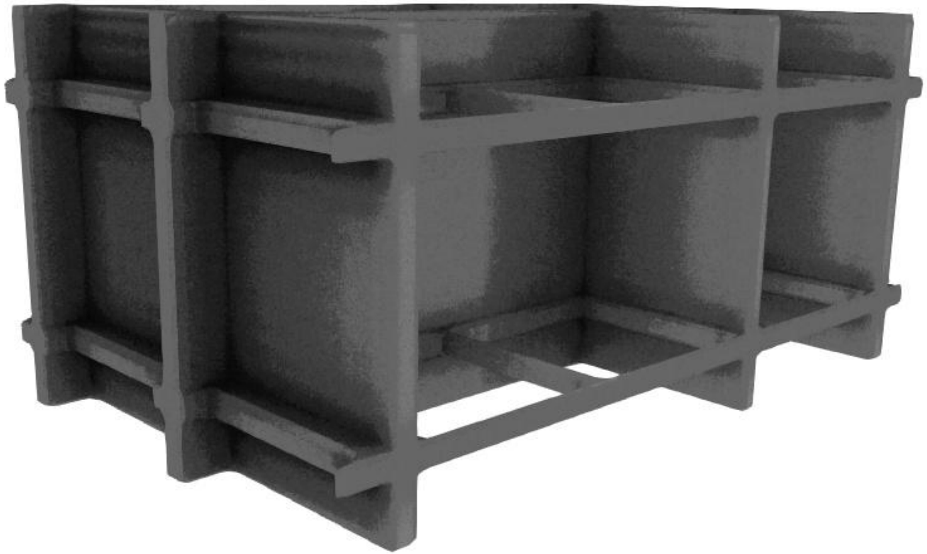
More Booleans

```
d = Box(pos)
c = fmod(pos * A, B)
subD = max(c.y,min(c.y,c.z))
subD = min(subD,cylinder(c))
subD = max(subD, Windows())
d = max(d, -subD)
```



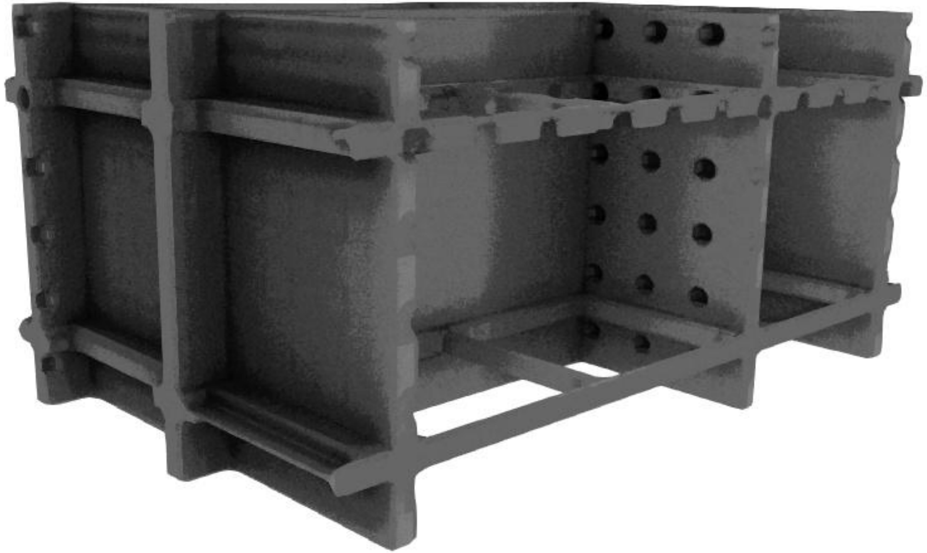
Repeated Booleans

```
d = Box(pos)
e = fmod(pos + N, M)
floorD = Box(e)
d = max(d, -floorD)
```



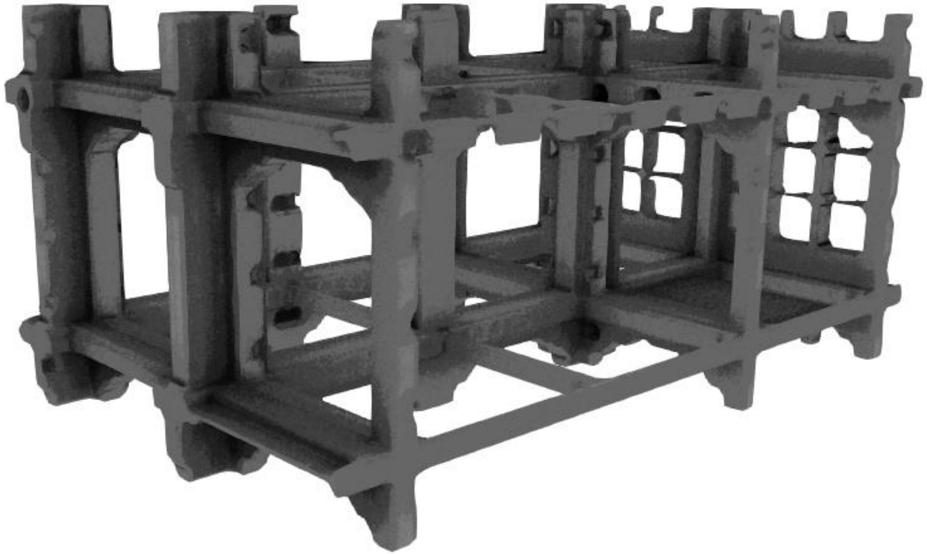
Cutting Holes

```
d = Box(pos)
e = fmod(pos + N, M)
floorD = Box(e)
floorD = min(floorD,holes())
d = max(d, -floorD)
```



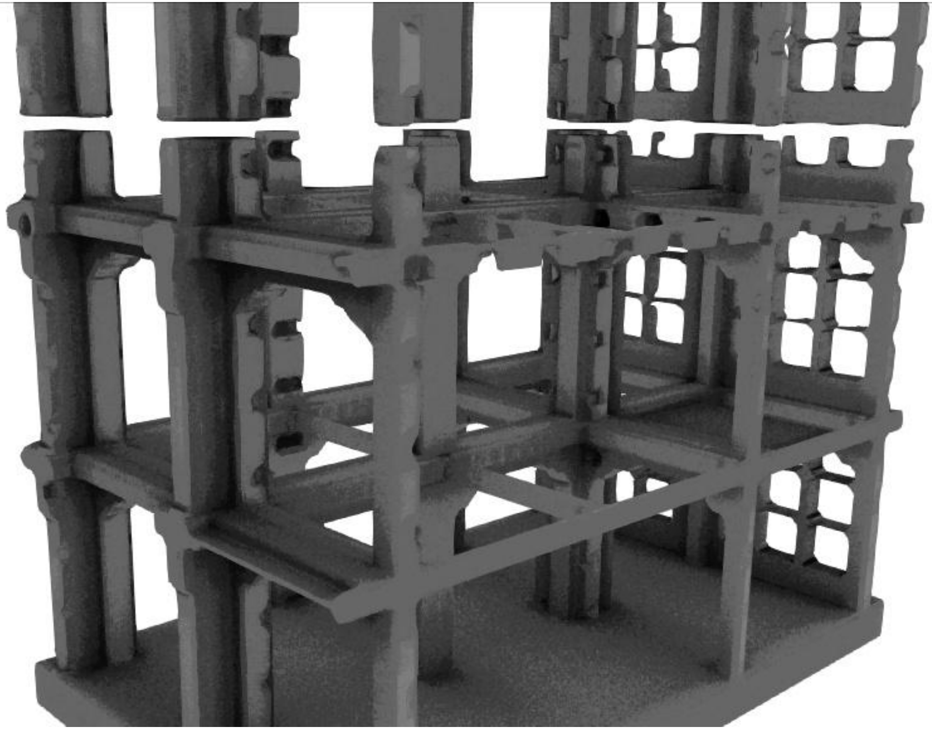
Combined Result

```
d = Box(pos)
c = fmod(pos * A, B)
subD = max(c.y,min(c.y,c.z))
subD = min(subD,cylinder(c))
subD = max(subD, Windows())
e = fmod(pos + N, M)
floorD = Box(e)
floorD = min(floorD,holes())
d = max(d, -subD)
d = max(d, -floorD)
```



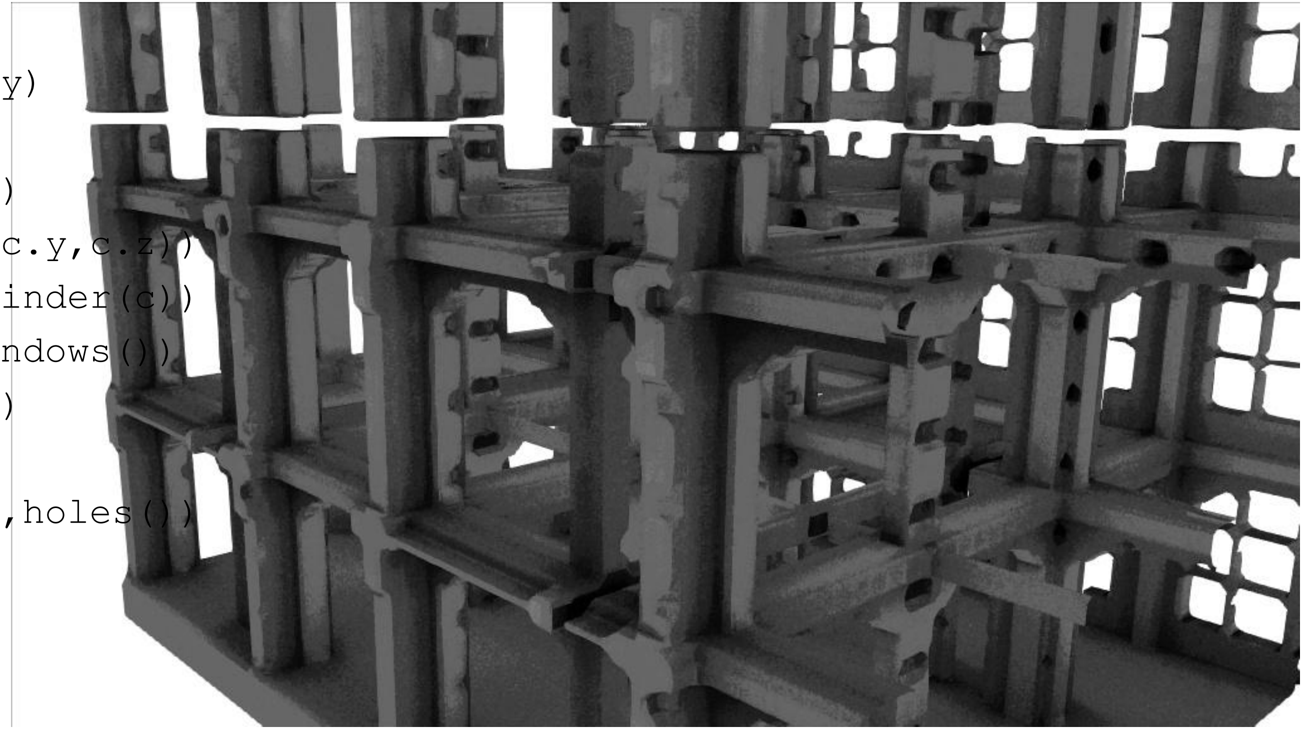
Repeating the Space

```
pos.y = frac(pos.y)
d = Box(pos)
c = fmod(pos * A, B)
subD = max(c.y,min(c.y,c.z))
subD = min(subD,cylinder(c))
subD = max(subD, Windows())
e = fmod(pos + N, M)
floorD = Box(e)
floorD = min(floorD,holes())
d = max(d, -subD)
d = max(d, -floorD)
```



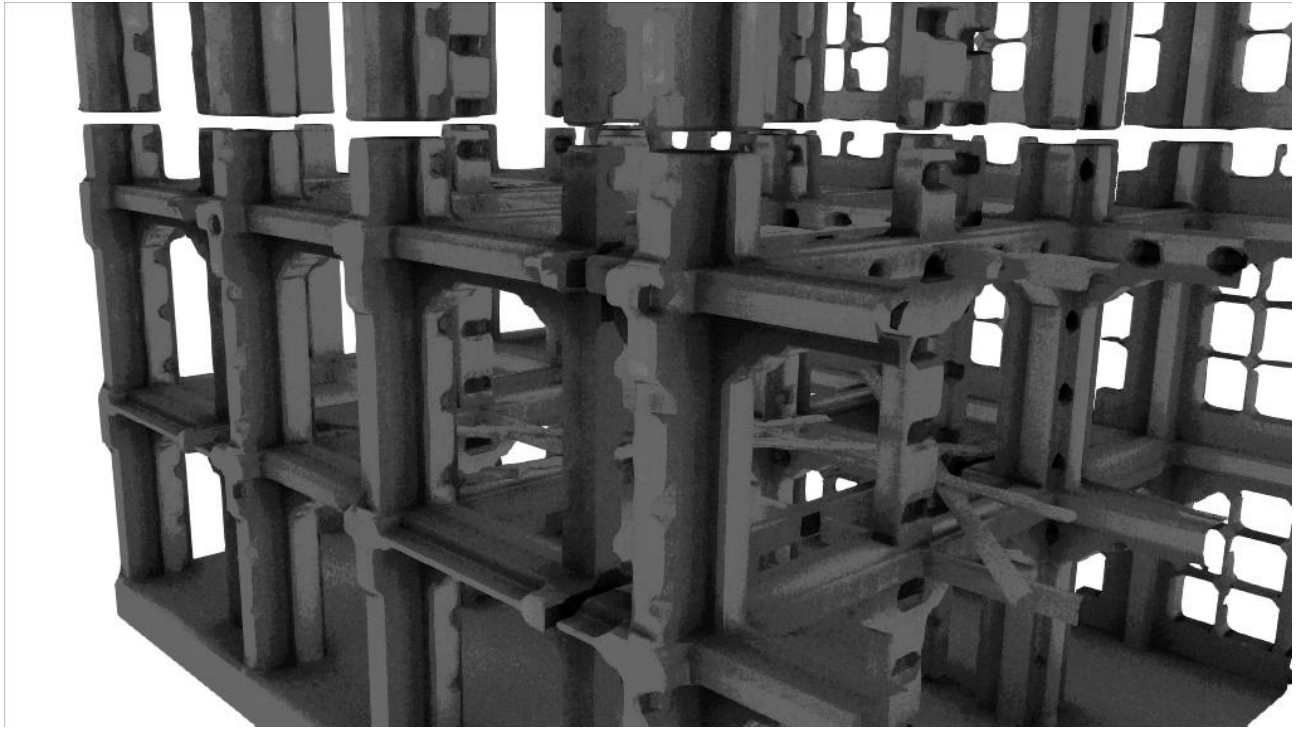
Repeating the Space

```
pos.xy = frac(pos.xy)
d = Box(pos)
c = fmod(pos * A, B)
subD = max(c.y,min(c.y,c.z))
subD = min(subD,cylinder(c))
subD = max(subD, Windows())
e = fmod(pos + N, M)
floorD = Box(e)
floorD = min(floorD,holes())
d = max(d, -subD)
d = max(d, -floorD)
```



Details

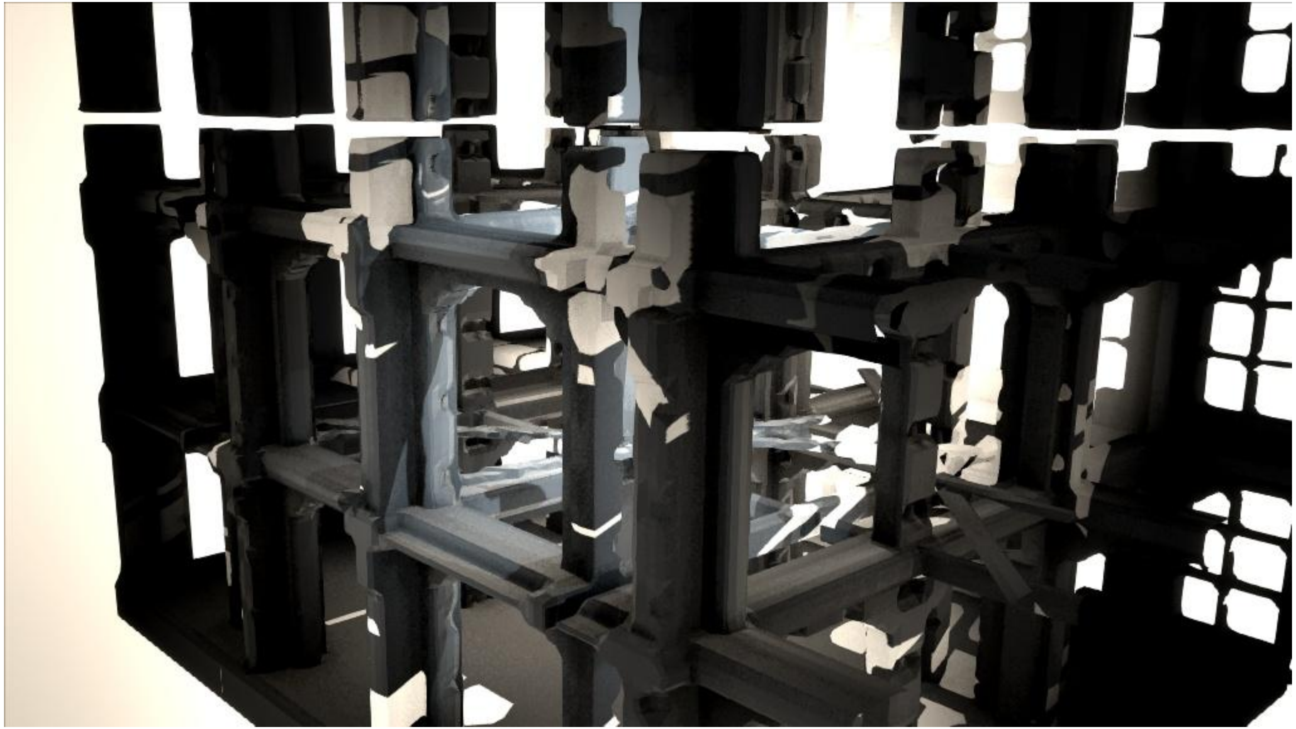
`AddDetails()`



Details

`DoLighting()`

`ToneMap()`



Details

`AddDeferredTexture()`

`AddGodRays()`

Details

`MoveCamera()`
`MakeLookGood()`

Ship It.

Procedural SDFs in Practice

- Generated scenes probably won’t replace 3D artists

Procedural SDFs in Practice

- Generated scenes probably won’t replace 3D artists

☹

Procedural SDFs in Practice

- Generated scenes probably won’t replace 3D artists
- Generated SDFs good proxies for real meshes
  - Code to combine a few primitives cheaper than art data
- Combine with artist-built meshes converted to SDFs
  - Boolean, modify, cut, distort procedurally

Video

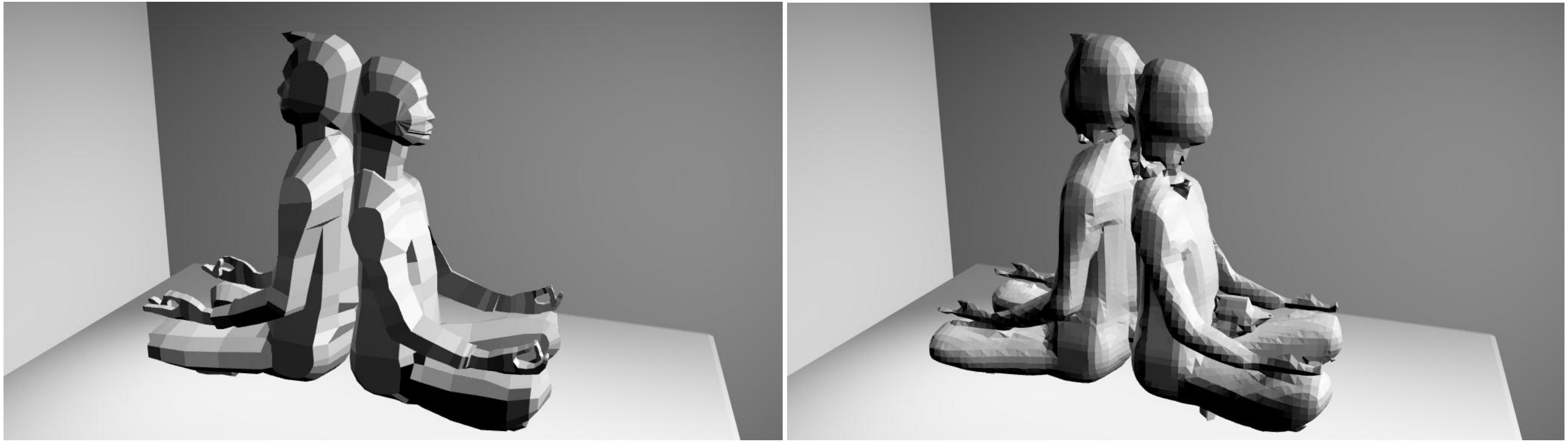
- (Video Removed)
- (It’s a cube morphing into a mesh. You know, just for fun etc.)

SDFs From Triangle Meshes

SDFs from Triangle Meshes

- Convert triangle mesh to SDF in 3D texture
  - 32^3 – 256^3 volume texture typical
  - SDFs **interpolate** well.. ← **bicubic interpolation**
  - .. Low resolution 3D textures still work well
  - Agnostic to poly count (except for processing time)
- Can often be done offline

SDFs from Triangle Meshes



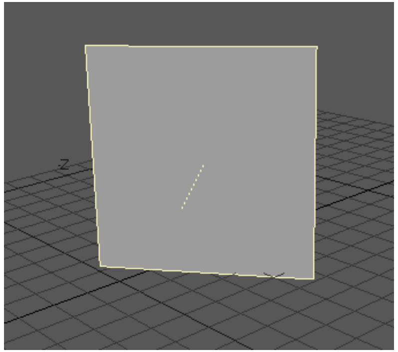
A mesh converted to a 64x64x64 SDF and polygonised.
It’s two people doing yoga, by the way.

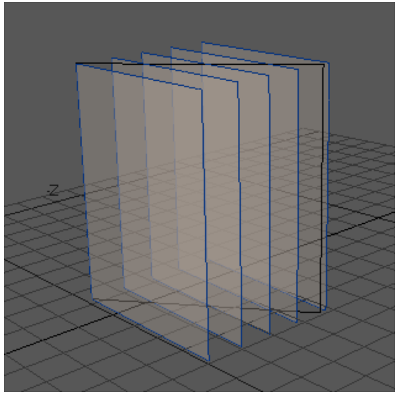
SDFs from Triangle Meshes

- Naïve approach?
  - Compute distance from every cell to every triangle
  - **Very slow** but accurate
- Voxelize mesh to grid, then sweep? ← **UGLY**
  - Sweep to compute signed distance from voxels to cells
  - Voxelization too inaccurate near surface..
  - ..But near-surface distance is important - **interpolation**
- Combine accurate triangle distance and sweep

Geometry Stages

- Bind 3D texture target
- VS transforms to SDF space
- Geometry shader replicates triangle to affected slices
    - Flatten triangle to 2D
    - Output positions as TEXCOORDs..
    - .. All 3 positions for each vertex





Pixel Shader Stage

- Calculates distance from 3D pixel to triangle
  - Compute closest position on triangle
  - Evaluate vertex normal using barycentric
- Evaluate distance sign using weighted normal
- Write signed distance to output color, distance to depth
- **Depth test** keeps closest distance

Post Processing Step

- Cells around mesh surface now contain accurate signed distance
- Rest of grid is empty
- Fill out rest of the grid in post process CS
- Fast Sweeping algorithm

Fast Sweeping

- Requires ability to read and write same buffer
- One thread per row
    - Thread R/W doesn’t overlap
    - No interlock needed
- Sweep forwards then backwards on same axis
- Sweep each axis in turn

```
d = maxPossibleDistance
for i = 0 to row length
      d += cellSize
      if(abs(cell[i]) > abs(d))
            cell[i] = d
      else
            d = cell[i]
```

SDFs from Particle Systems

SDFs From Particle Systems

- Naïve: treat each particle as a sphere
  - Compute min distance from point to particles
- Better: use metaball blobby equation
  - **Density(P) = Sum[$(1 - (r^2/R^2))^3$]** for all particles
    - R : radius threshold
    - r : distance from particle to point P
- Problem: checking all particles per cell

Evaluating Particles Per Cell

- Bucket sort particles into grid cells in CS
- Evaluate a kernel around each cell
  - Sum potentials from particles in neighbouring cells
  - **9x9x9** kernel typical
  - (729 cells, containing multiple particles per cell, evaluated for ~2 million grid cells)
- Gives accurate result .. glacially
  - > **200**ms on Geforce 570

Evaluating Particles, Fast

- Render single points into grid
  - Write out particle position with additive blend
  - Sum particle count in alpha channel
- Post process grid
  - Divide by count: get average position of particles in cell
- Evaluate potentials with kernel - **grid cells only**
  - Use grid cell average position as proxy for particles

Evaluating Particles, **Faster**

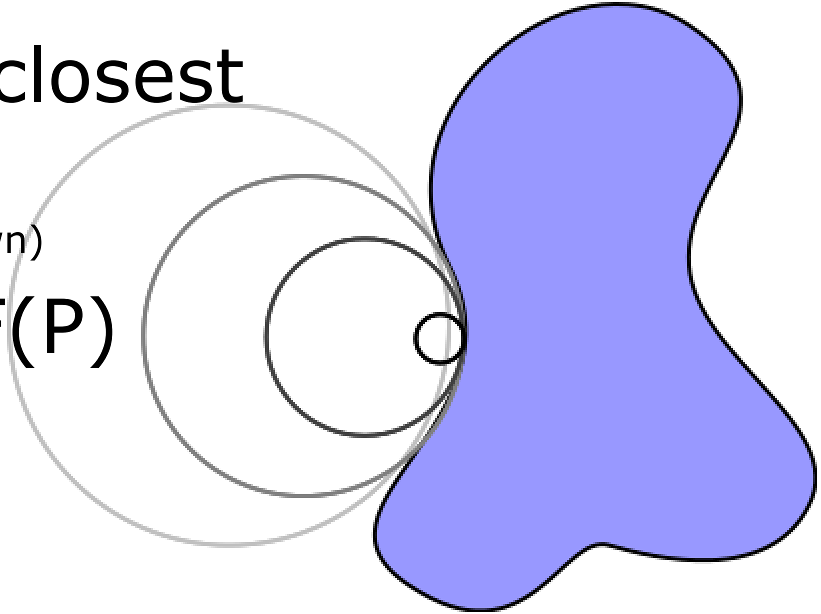
- Evaluating potentials accurately far too slow
  - Summing e.g. 9x9x9 cell potentials for each cell..
  - Still > 100 ms for our test cases
- Use **separable blur** to spread potentials instead
  - Not quite 100% accurate.. But **close enough**
  - Calculate blur weights with potential function to at least feign correctness
- Hugely faster - < **2** ms

Visualising Distance Fields

Ray Tracing & Polygonisation

Ray Casting

See: *ray marching; sphere tracing*



- SDF(P) = Distance to closest point on surface
    - (Closest point's actual location not known)
- Step along ray by SDF(P) until SDF(P)~0
- Skips empty space!

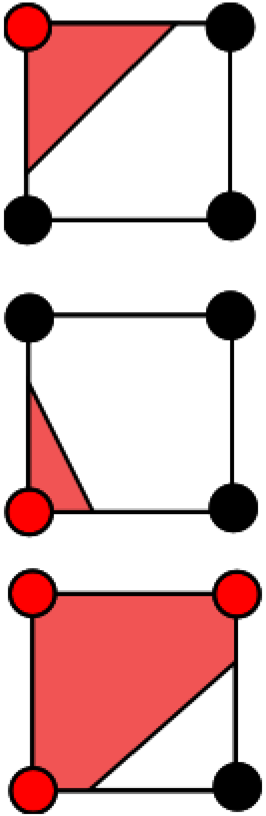
Ray Casting

- Accuracy depends on iteration count
- Primary rays require high accuracy
  - 50-100 iterations -> **slow**
  - Result is transitory, view dependent
- Useful for secondary rays
  - Can get away with fewer iterations
- Do something else for primary hits

Polygonisation / Meshing

- Generate triangle mesh from SDF
- Rasterise as for any other mesh
  - Suits 3D hardware
  - Integrate with existing render pipeline
  - **Reuse mesh** between passes / frames
  - Speed not dependent on **screen resolution**
- Use **Marching Cubes**

Marching Cubes In One Slide



- Operates on a discrete grid
- Evaluate field F() at 8 corners of each cubic cell
  - Generate sign flag per corner, OR together
- Where sign(F) changes across corners, triangles are generated
  - 5 per cell max
- Lookup table defines triangle pattern

Marching Cubes Issues

- Large number of grid cells
  - 128x128x128 = **2 million cells**
  - Only process whole grid when necessary
- Triangle count varies hugely by field contents
  - Can change radically every frame
  - Upper bound very large: -> size of grid
  - Most cells empty: actual output count **relatively small**
- Traditionally implemented on **CPU**

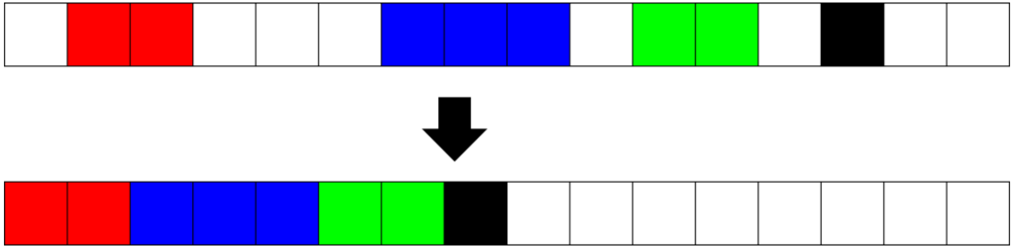
Geometry Shader Marching Cubes

- CPU submits a large, empty draw call
  - One point primitive per grid cell (i.e. a lot)
  - VS minimal: convert SV_VertexId to cell position
- GS evaluates marching cubes for cell
  - Outputs 0 to 5 triangles per cell
- Far too slow: **10**ms - **150**ms (128^3 grid, architecture-dependent)
  - Work per GS instance varies greatly: poor parallelism
  - Some GPU architectures handle GS very badly

Stream Compaction on GPU

Stream Compaction

- Take a sparsely populated array



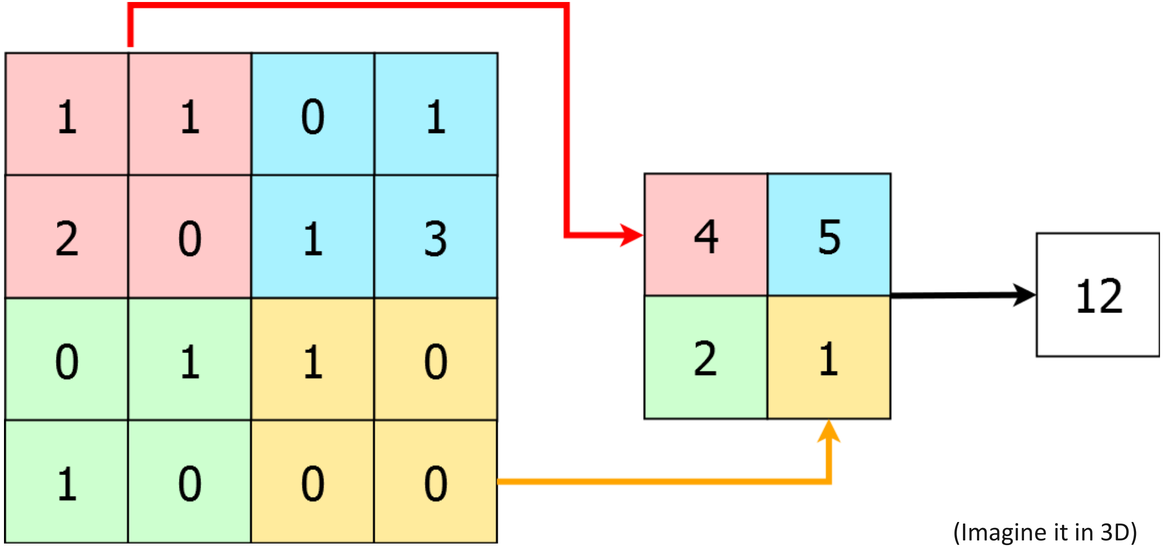
- Push all the filled elements together
  - Remember count & offset mapping
- Now only have to process filled part of array

Stream Compaction

- Counting pass - parallel reduction
    - Iteratively halve array size (like mip chain)
    - Write out the sum of the count of parent cells
    - Until final step reached: 1 cell, the total count
- Offset pass - iterative walk back up
    - Cell offset = parent position + sibling positions
- Histopyramids: stream compaction in 3D

Histopyramids

- Sum down mip chain in blocks



1	1	0	1
2	0	1	3
0	1	1	0
1	0	0	0

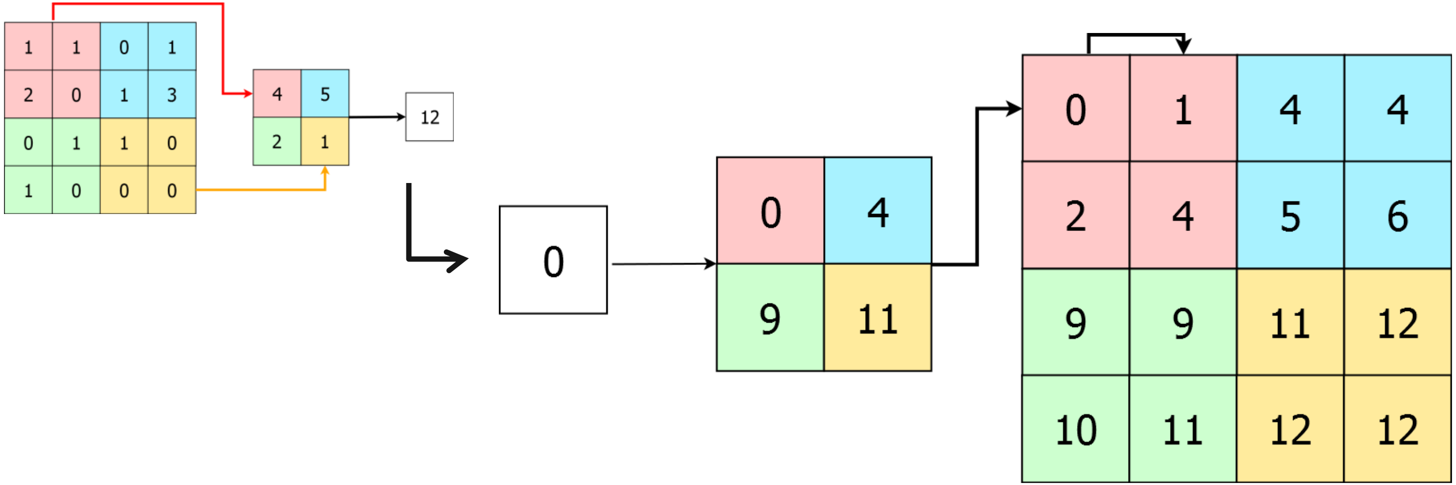
4	5
2	1

12

(Imagine it in 3D)

Histopyramids

- Count up from base to calculate offsets



1	1	0	1
2	0	1	3
0	1	1	0
1	0	0	0

4	5
2	1

0	4
9	11

0	1	4	4
2	4	5	6
9	9	11	12
10	11	12	12

Histopyramids In Use

- Fill grid volume texture with active mask
  - 0 for empty, 1 for active
- Generate counts in mip chain downwards
- Use 2nd volume texture for cell locations
  - Walk up the mip chain

Compaction In Action

- Use histopyramid to **compact active cells**
  - Active cell **count** now known too
- GPU dispatches drawcall only for # active cells
  - Use DrawInstancesIndirect
- GS determines grid position from cell index
  - Use histopyramid for this
- Generate marching cubes for cell in GS

Compaction Reaction

- Huge improvement over brute force
  - ~5 ms – down from 11 ms
  - Greatly improves parallelism
  - Reduced draw call size
- Geometry still generated in GS
  - Runs again for each render pass
  - No indexing / vertex reuse

Geometry Generation

Generating Geometry

- Wish to pre-generate geometry (no GS)
  - Reuse geometry between passes; allow indexed vertices
- First generate active vertices
  - Intersection of grid edges with 0 potential contour
  - Remember vertex index per grid edge in lookup table
  - Vertex count & locations still vary by potential field contents
- Then generate indices
  - Make use of the vertex index lookup

Generating Vertex Data

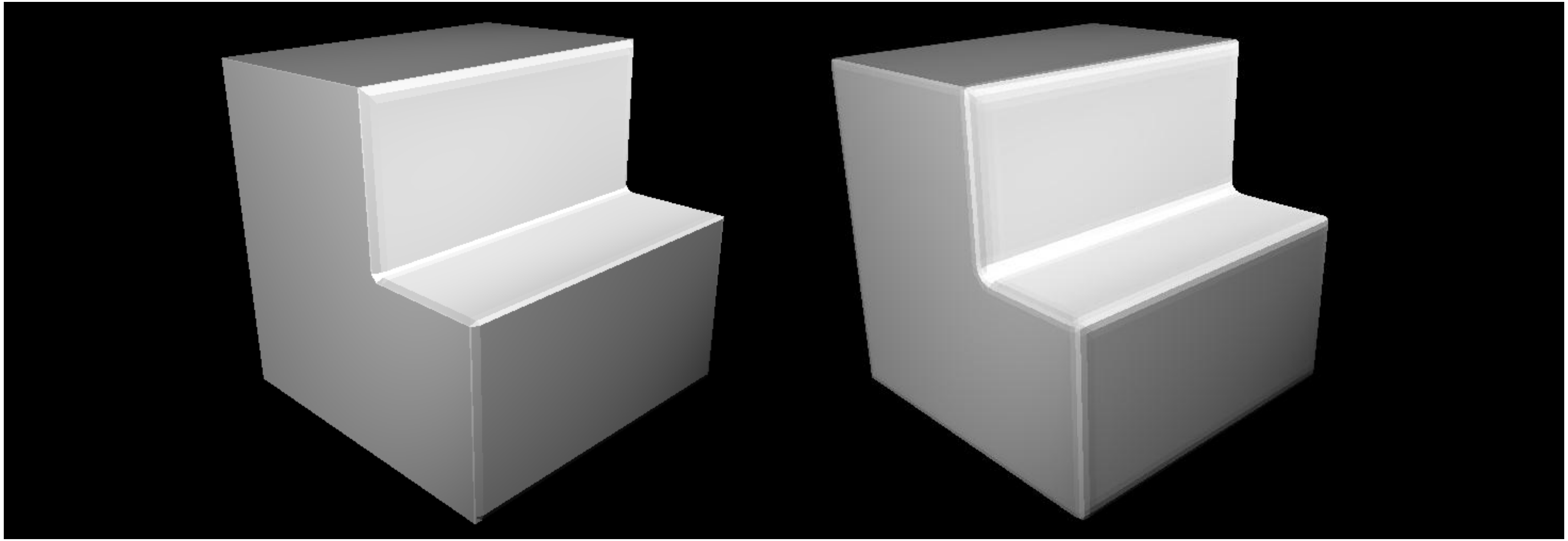
- Process potential grid in CS
  - One cell per thread
  - Find active edges in each cell
- Output vertices per cell
  - IncrementCounter() on vertex buffer
    - Returns current num vertices written
  - Write vertex to end of buffer at current counter
  - Write counter to edge index lookup: scattered write
- Or use 2nd histopyramid for vertex data instead

Generating Geometry

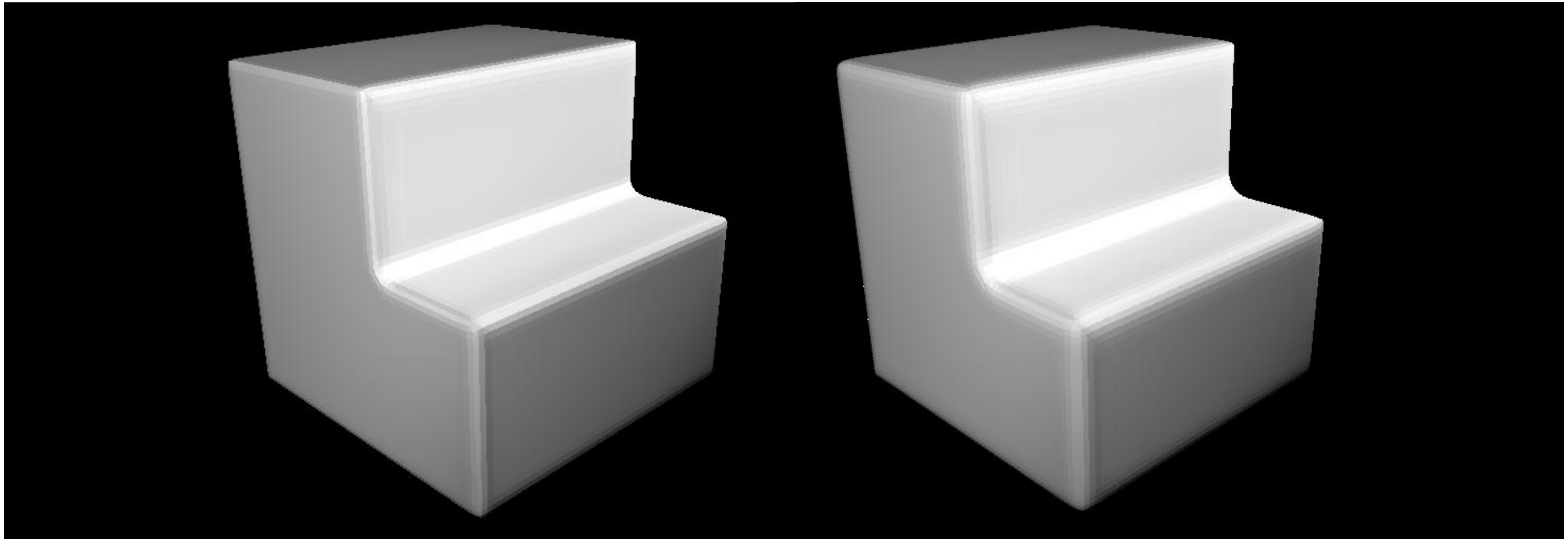
- Now generate index data with another CS
    - Histopyramid as before..
    - .. But use edge index grid lookup to locate indices
    - DispatchIndirect to limit dispatch to # active cells
- Render geom: DrawIndexedInstancedIndirect
    - GPU draw call: index count copied from histopyramid
- No GS required! Generation can take just **2**ms

Meshing Improvements

Smoothing



Smoothing More



Smoothing

- Laplacian smooth
  - Average vertices along edge connections
    - Key for improving quality of fluid dynamics meshing
- Must know vertex edge connections
  - Generate from index buffer in post process

Bucket Sorting Arrays

- Need to bucket elements of an array?
  - E.g. Spatial hash; particles per grid cell; triangles connected to each vertex
- Each bucket has varying # elements
- Don’t want to over-allocate buckets
  - Allocate only # elements in array

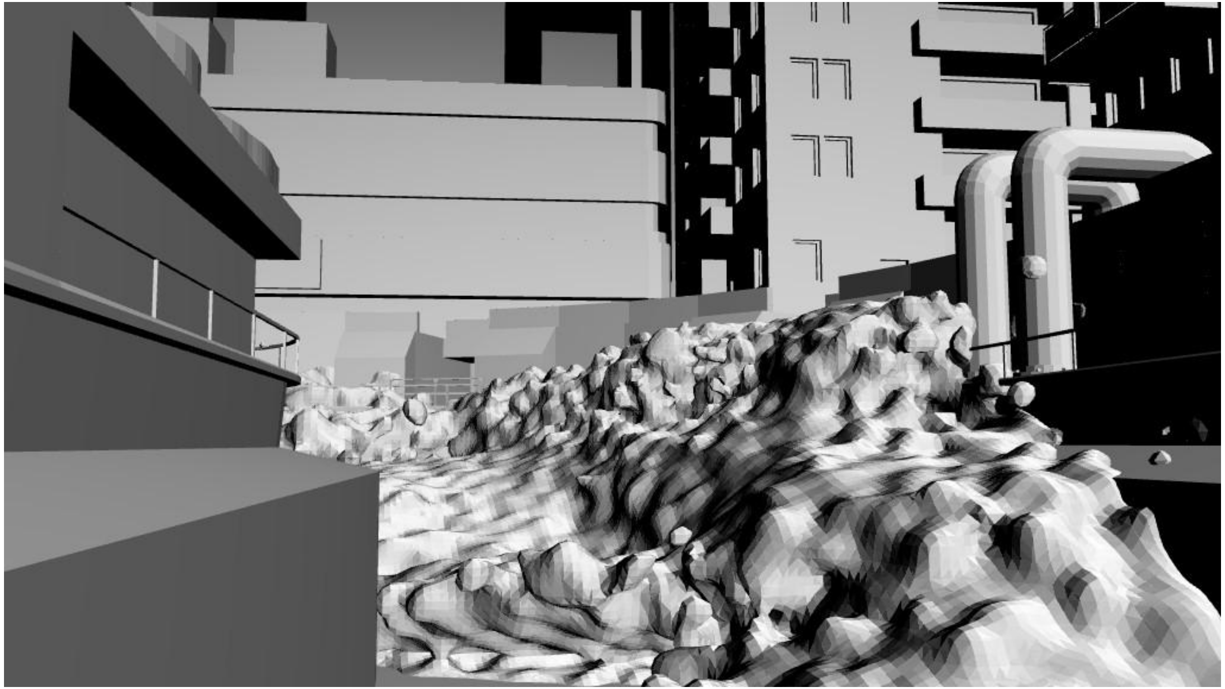
Counting Sort

- Use Counting Sort
- Counting pass – **count** # elements per bucket
  - Use atomics for parallel op – InterlockedAdd()
- Compute **Parallel Prefix Sum**
  - Like a 1d histopyramid.. See CUDA SDK
  - Finds **offset** for each bucket in element array
- Then assign elements to buckets
  - Reuse counter buffer to track idx in bucket

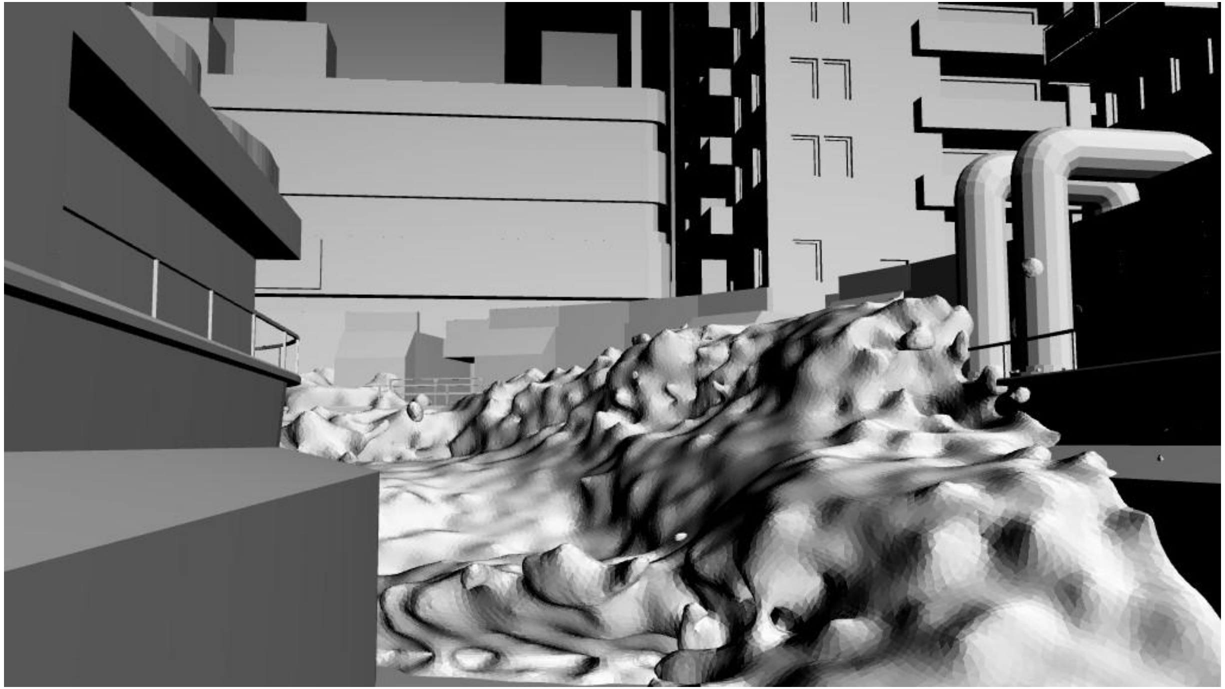
Smoothing Process

- Use **Counting Sort:** bucket **triangles** per vertex
- Post-process: determine **edges** per vertex
- Smooth vertices
  - (Original Vertex * 4 + Sum[Connected Vertices]) / (4 + Connected Vertex Count)
  - Iterate smooth process to increase smoothness

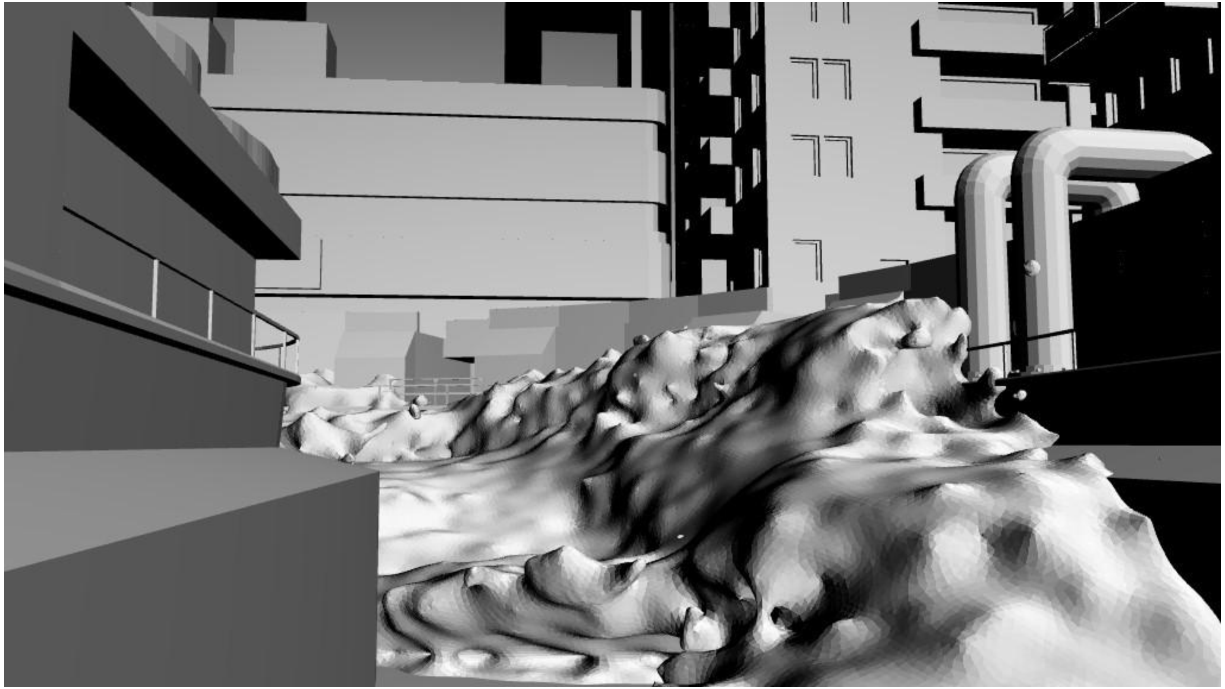
0 Smoothing Iterations



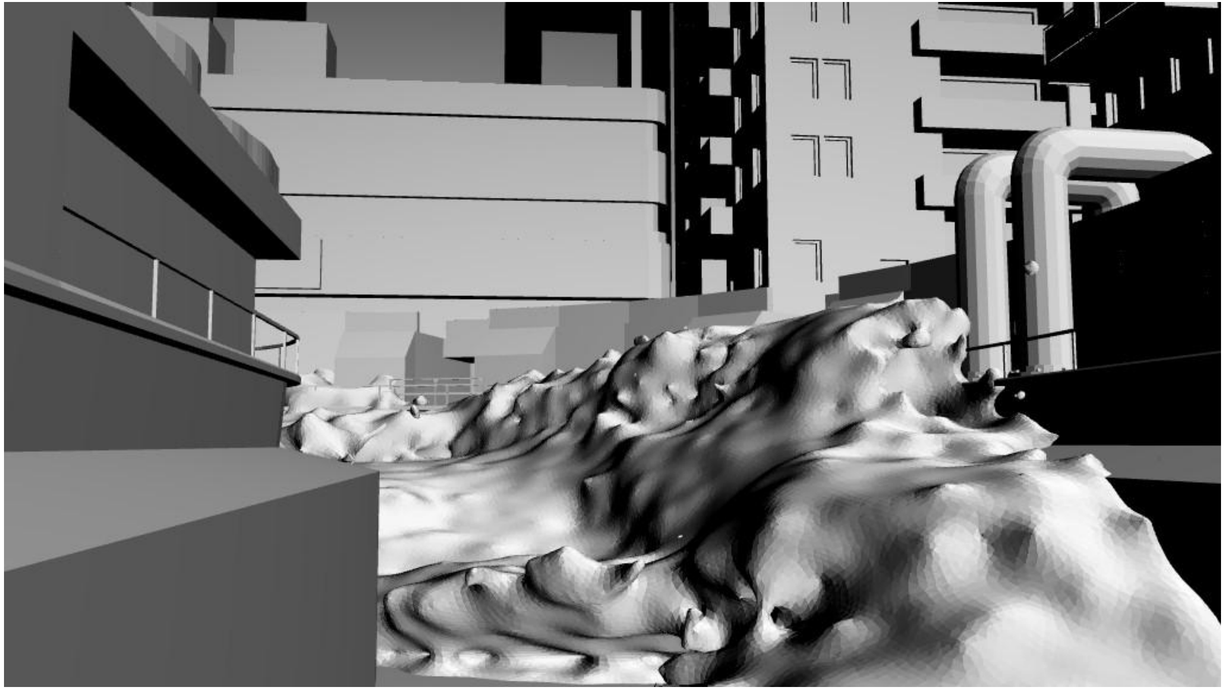
4 Smoothing Iterations



8 Smoothing Iterations



16 Smoothing Iterations



Subdivision, Smooth Normals

- Use existing vertex connectivity data
- Subdivision: split edges, rebuild indices
  - 1 new vertex per edge
  - 4 new triangles replace 1 old triangle
- Calc smooth vertex normals from final mesh
  - Use vertex / triangle connectivity data
  - Average triangle face normals per vertex
  - Very fast – minimal overhead on total generation cost

Performance

- Same scene, **128**^3 grid, **Geforce 570**
- Brute force GS version: **11** ms **per pass**
  - No reuse – shadowmap passes add 11ms each
- Generating geometry in CS: **2** ms + **0.4** ms per pass
  - 2ms to generate geometry in CS; 0.4ms to render it
  - Generated geometry reused between shadow passes

Video

- (Video Removed)
- (A tidal wave thing through a city. It was well cool!!!!!1)

Wait, Was That Fluid Dynamics?

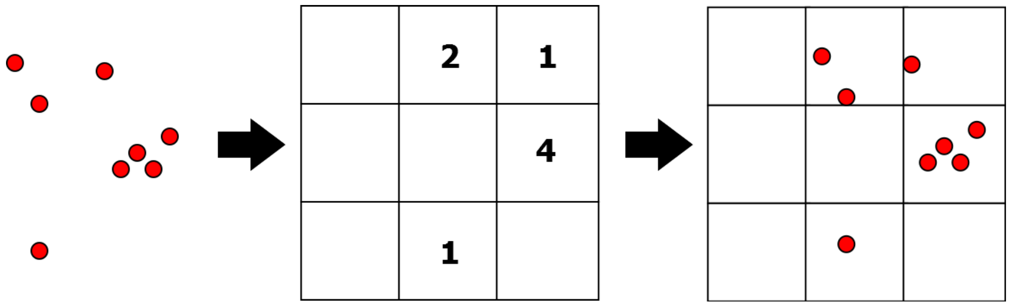
Yes, It Was.

Smoothed Particle Hydrodynamics

- Solver works on **particles**
  - Particles represent **point samples** of fluid in space
- Locate local neighbours for each particle
  - Find all particles inside a particle’s smoothing radius
  - **Neighbourhood search** – can be expensive
- Solve fluid forces between particles within radius
- We use Compute for most of this

Neighbourhood Search

- Spatially bucket particles using spatial hash
- Return of **Counting Sort** - with a histopyramid
    - In this case: hash is quantised 3D position
    - Bucket particles into hashed cells



SPH Process – Step by Step

- Bucket particles into cells
- Evaluate all particles..
- Find particle neighbours from cell structure
  - Must check all nearby cells inside search radius too
- Sum forces on particles from all neighbours
  - Simple equation based on distance and velocities
- Return new acceleration

SPH Performance

- Performance depends on # neighbours evaluated
  - Determined by cell granularity, particle search radius, number of particles in system, area covered by system
- Favour small cell granularity
  - Easier to reduce # particles tested at cell level
- Balance particle radius by hand
  - Smoothness vs performance

SPH Performance

- In practice this is still far, far **too slow** (>200ms)
  - Can check > 100 cells, too many particle interactions
- So we **cheat**..
  - Average particle positions + velocities in each cell
  - Use average value for particles vs distant cells
  - Force vectors produced **close enough** to real values..
- Only use real particle positions for close cells

Illumination

The Rendering Pipeline of the Future

Rendering Pipeline of the Future

- **Primary** rays are **rasterised**
  - Fast: rasterisation still faster for typical game meshes
  - Use for camera / GBuffers, shadow maps
- **Secondary** rays are **traced**
  - Use GBuffers to get starting point
  - Global illumination / ambient occlusion, reflections
  - Paths are complex – bounce, scatter, diverge
  - Needs full scene knowledge – hard for rasterisation
  - Tend to need less accuracy / sharpness..

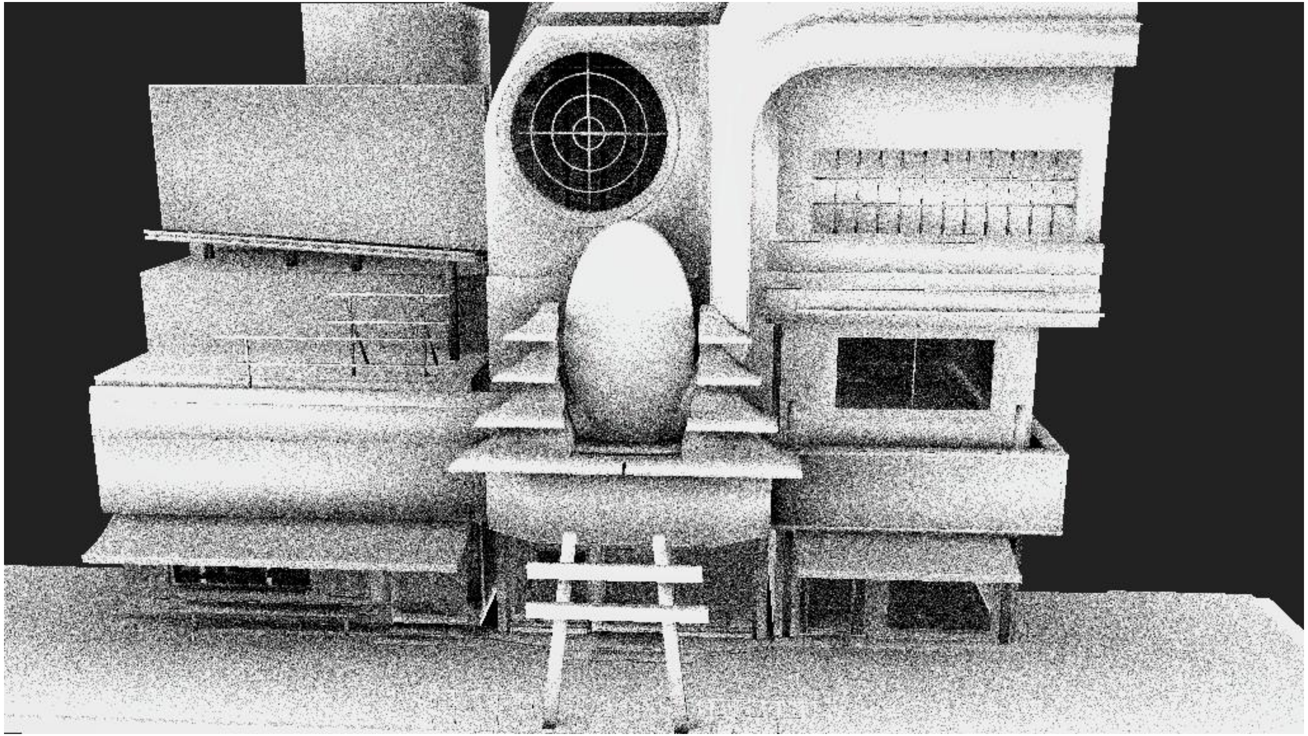
Ambient Occlusion Ray Tracing

- Cast many random rays out from surface
  - Monte-Carlo style
- AO result = % of rays that reach sky
- Slow..
  - Poor ray coherence
  - Lots of rays per pixel needed for good result
- Some fakes available
  - SSAO & variants – largely horrible.. ☺

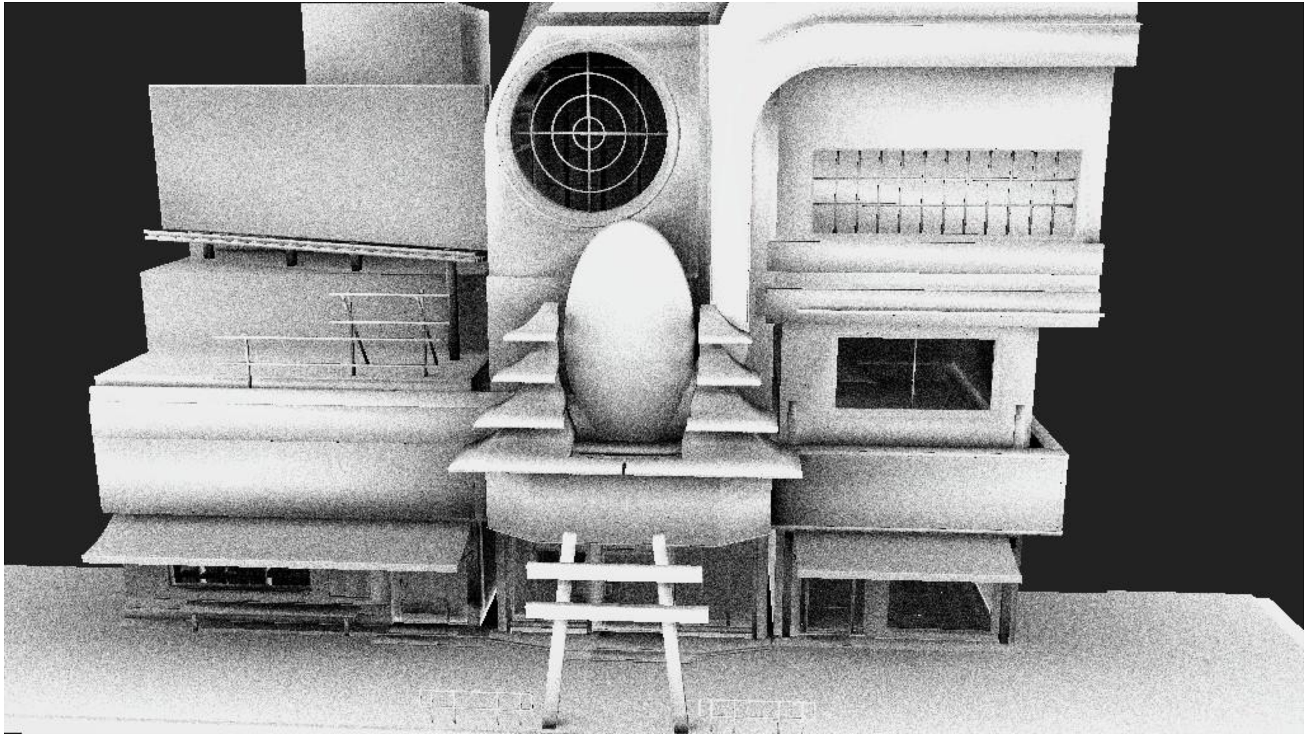
Ambient Occlusion with SDFs

- Raytrace SDFs to calculate AO
- Accuracy less important (than primary rays)
  - Less SDF iterations – < 20, not 50-100
- Limit ray length
- We don't really "ray cast"..
  - Just sample multiple points along ray
  - Ray result is a function of SDF distance at points

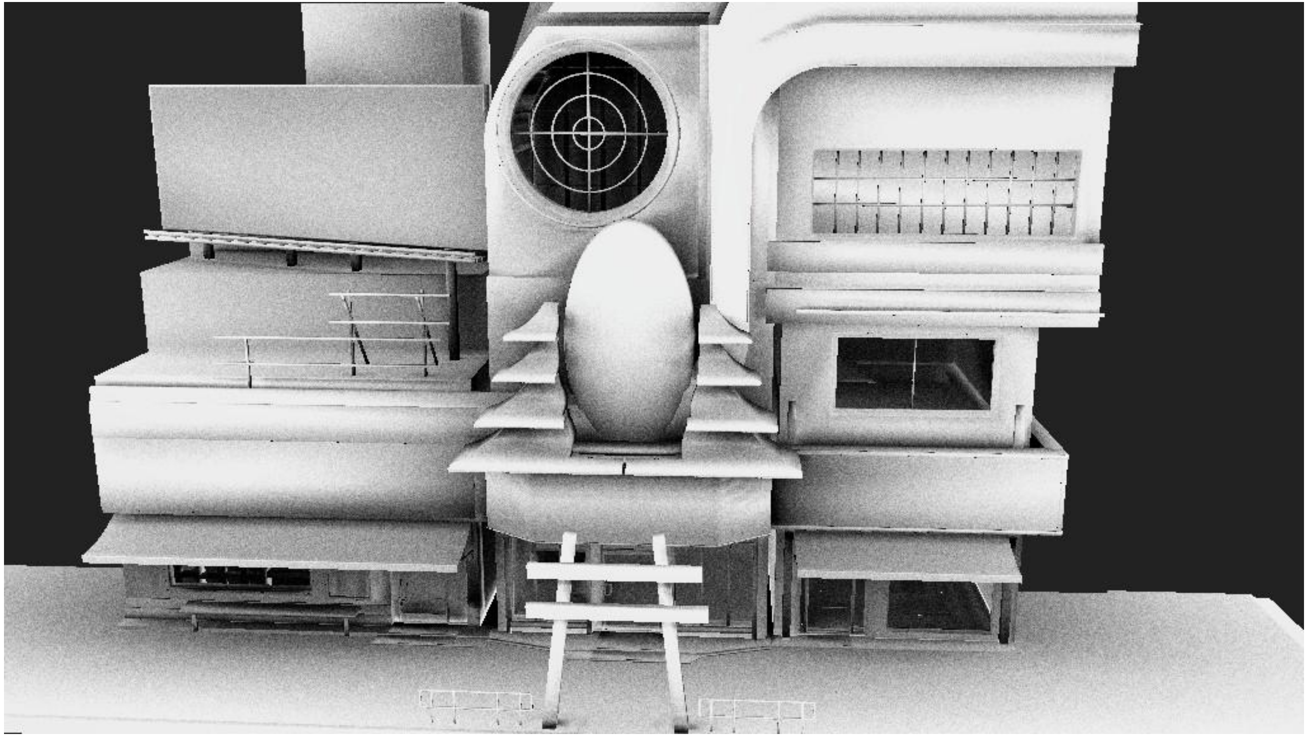
4 Rays Per Pixel



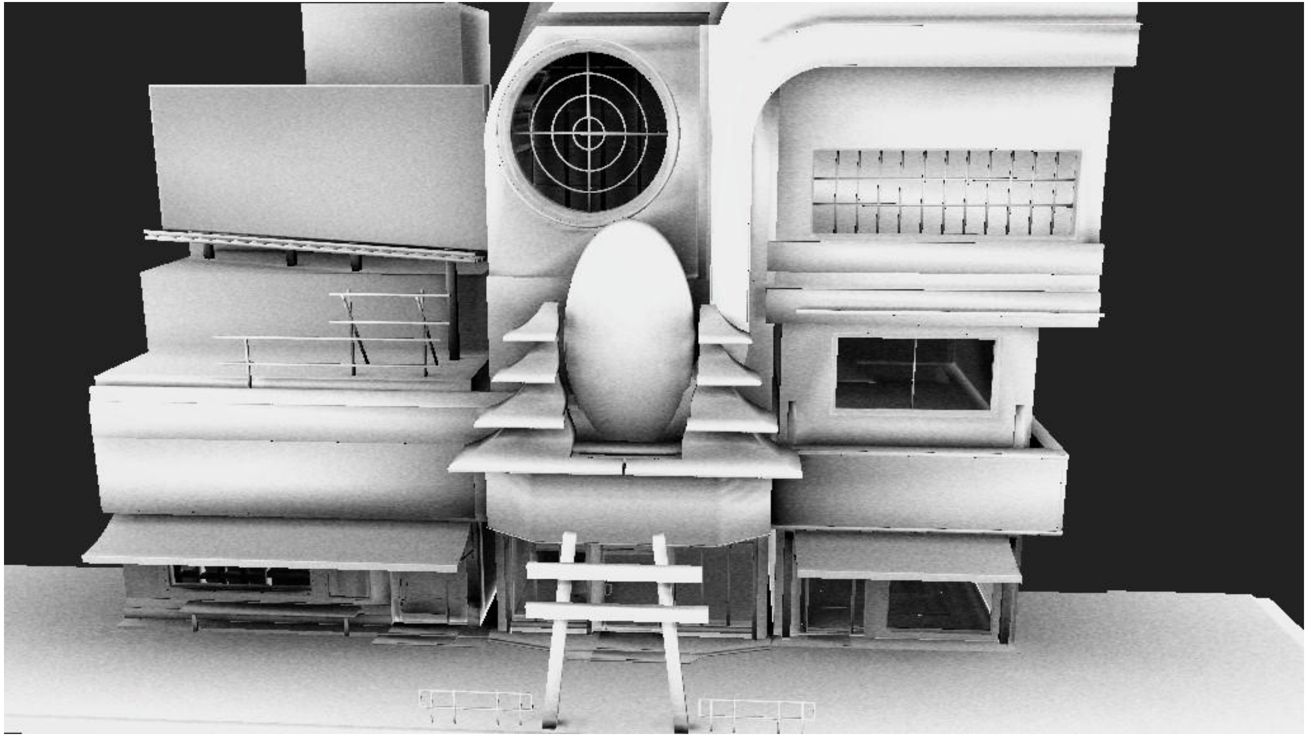
16 Rays Per Pixel



64 Rays Per Pixel



256 Rays Per Pixel



Ambient Occlusion Ray Tracing

- Good performance: 4 rays; Quality: 64 rays
- Try to plug quality/performance gap
- Could bilateral filter / blur
  - Few samples, smooth results spatially (then add noise)
- Or use **temporal reprojection**
  - Few samples, refine results **temporally**
  - **Randomise** rays differently every frame

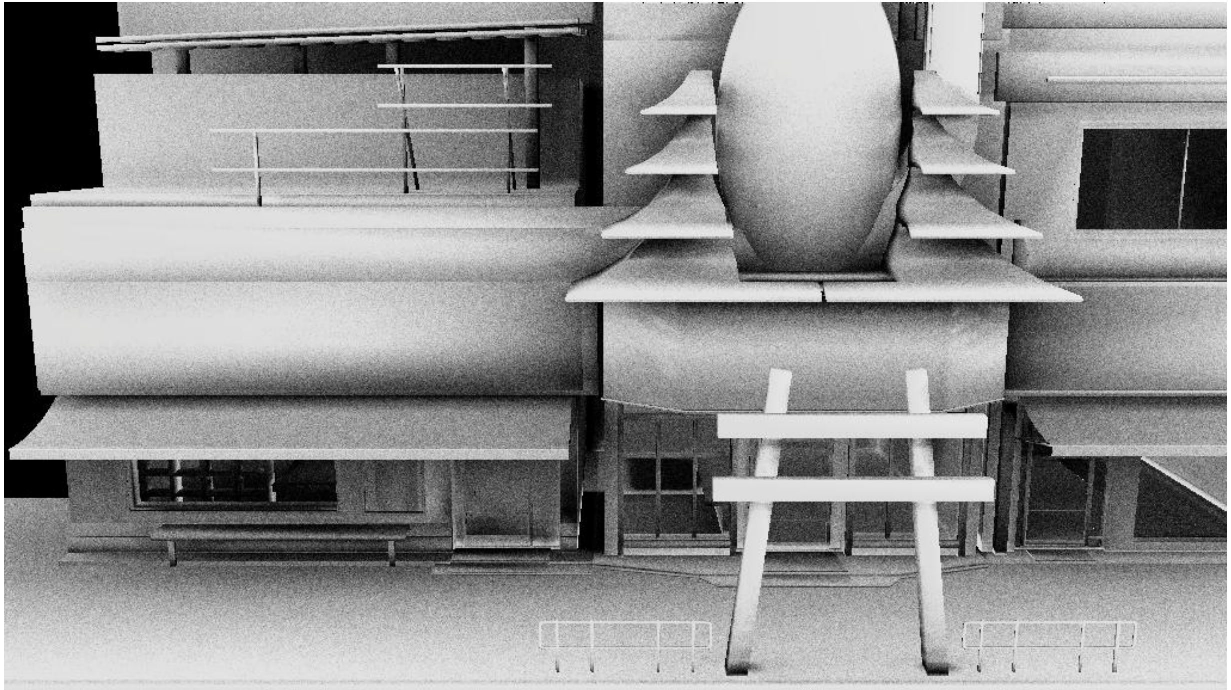
Temporal Reprojection

- Keep previous frame’s data
  - Previous result buffer, normals/depths, view matrix
- **Reproject** current frame → previous frame
  - Current view position * view inverse * previous view
  - Sample previous frame’s result, blend with current
  - Reject sample if normals/depths differ too much
- Problem: **rejected samples** / holes

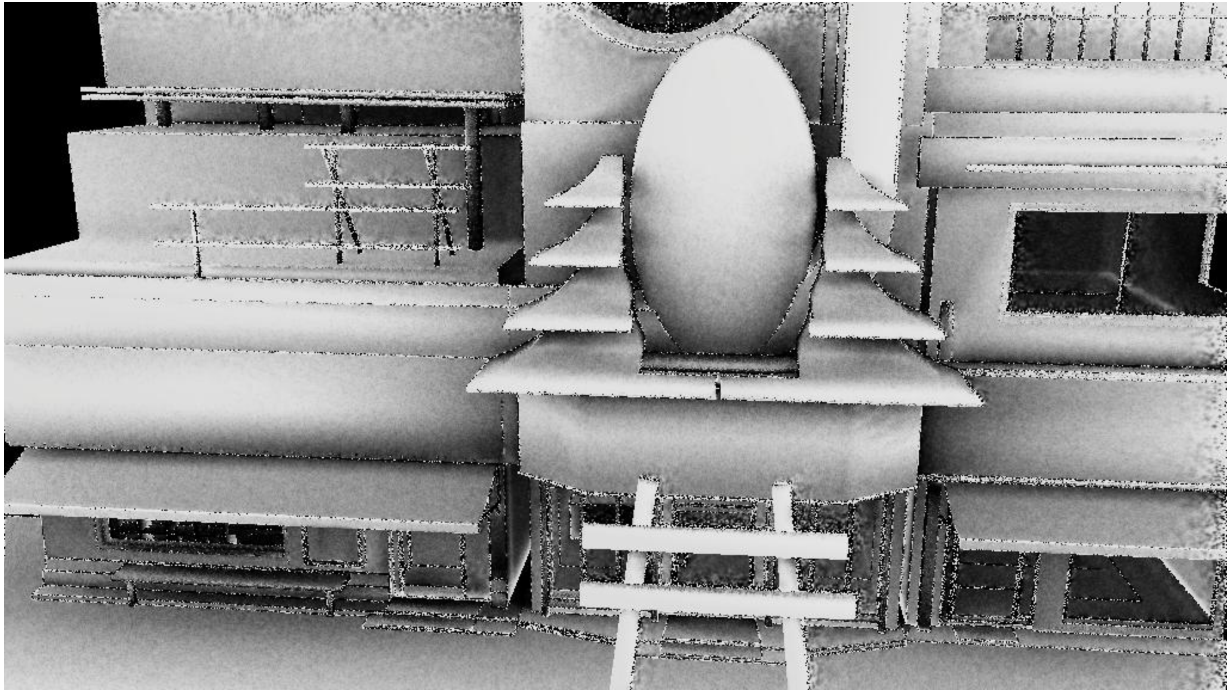
Video

- (Video Removed)
- (Basically it looks noisy, then temporally refines, then when the camera moves you see holes)

Temporal Reprojection: Good



Temporal Reprojection: Holes



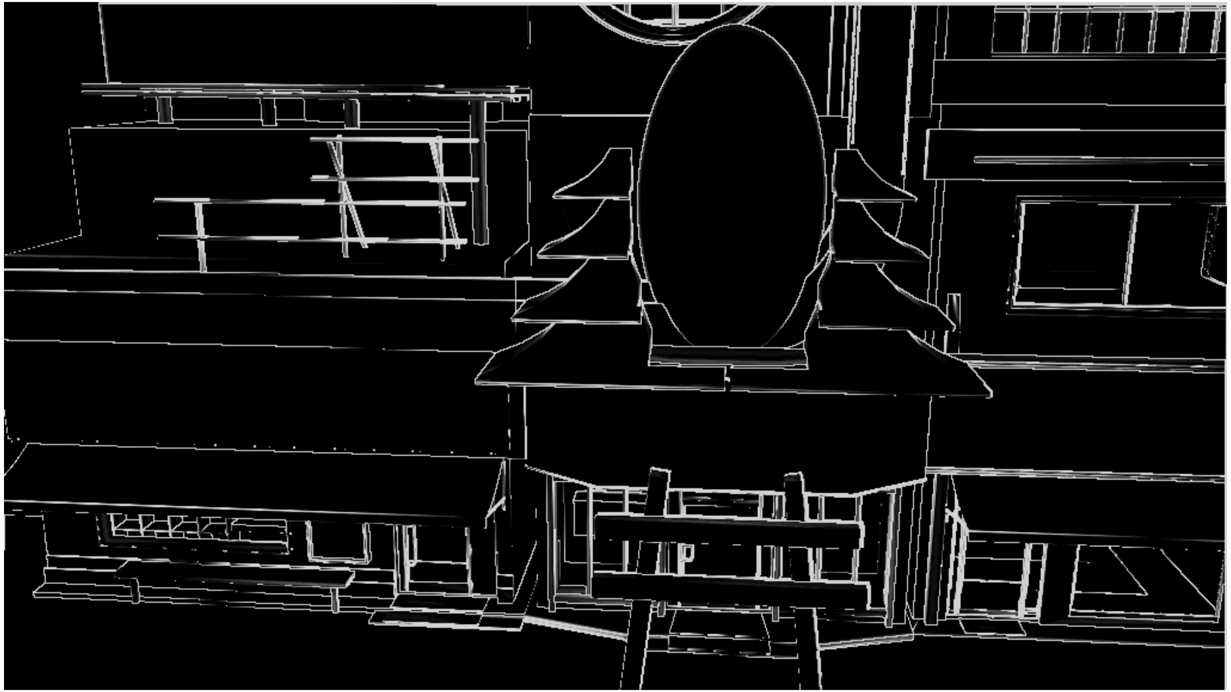
Hole Filling

- Reprojection works if you can fill holes nicely
- Easy to fill holes for AO: just **cast more rays**
  - Cast **16** rays for pixels in holes, **1** for the rest
- Adversely affects performance
  - Work between local pixels differs greatly
  - CS thread groups **wait** on **longest thread**
  - Some threads take **16x** longer than others to complete

Video

- (Video Removed)
- (It looks all good cos the holes are filled)

Rays Per Thread



Hole Filling

- Solution: **balance** rays across threads in CS
- 16x16 pixel tiles: 256 threads in group
- Compute & sum up required rays in tile
  - 1 pixel per thread
  - 1 for reprojected pixels; 16 for hole pixels
- Spread ray evaluation across cores evenly
  - N rays per thread

Rays Per Thread - Tiles

Video

- (Video Removed)
- (It still looks all good cos the holes are filled, by way of proof I’m not lying about the technique)

Performance

- 16 rays per pixel: **30** ms
- 1 ray per pixel, reproject: **2** ms
- 1 + 16 in holes, reproject: **12** ms
- 1 + 16 rays, load balanced tiles: **4** ms
  - ~ 2 rays per thread typical!

Looking Forward

Looking Forward

- Multiple representations of same world
    - Geometry + SDFs
    - Rasterise them
    - Trace them
    - Collide with them
- → World can be more dynamic.

http://directtovideo.wordpress.com

Thanks

- Jani Isoranta, Kenny Magnusson for 3D
- Angeldawn for the Fairlight logo
- Jussi Laakonen, Chris Butcher for actually making this talk happen
- SCEE R&D for allowing this to happen
- Guillaume Werle, Steve Tovey, Rich Forster, Angelo Pesce, Dominik Ries for slide reviews

References

- *High-speed Marching Cubes using Histogram Pyramids*; Dyken, Ziegler et al.
- *Sphere Tracing: a geometric method for the antialiased ray tracing of implicit surfaces*; John C. Hart
- *Rendering Worlds With Two Triangles*; Inigo Quilezles
- *Fast approximations for global illumination on dynamic scenes;* Alex Evans