GPU-based clay simulation and ray-tracing tech in Claybook

Sebastian Aaltonen
Co-founder of Second Order
Introduction

- **Sebastian Aaltonen**
  - Ex-Ubisoft senior lead programmer
  - 20 years of 3d programming experience

- **Second Order**
  - Formed two years ago
  - Two employees (me and Sami)
  - We target PC and consoles
  - Claybook is our first game
Topics

- Claybook Overview
- Signed Distance Fields (SDF)
- Raytracing Signed Distance Fields
- Clay and Fluid Simulation
- Async Compute
- Integration to Unreal Engine 4
Claybook Overview

● Clay simulation game
● Fully destructible environment
● User generated content
● PC (Steam), Xbox One (X) and PS4 (Pro)
● Steam Early Access & Xbox Game Preview
Claybook Overview, cont

- Clay modeled as signed distance fields (SDF)
  - Both world and characters are SDF based
- Physics & fluid simulation running on GPU
- No baked lighting, AO or shadows
  - Everything must be real time
Claybook Trailer

https://www.youtube.com/watch?v=Q8quiLN7n04
Signed Distance Fields (SDF)

- **SDF(P)** = signed distance to nearest surface at P
- Analytic distance functions
  - Popular in demoscene productions
  - Huge shader. Lots of math. No data
- Volume texture
  - Store distance function. Trilinear filter
  - We use volume texture with mip maps
World SDF

- Resolution = 1024x1024x512
- Format = 8 bit signed
- Size = 586 MB (5 mip levels)
- Distance of [-4, +4] voxels
  - 256 values / 8 voxels → 1/32 voxel precision
  - Max step distance (world space) doubled per mip level
SDF Brushes

- Brush = Small offline baked volume texture
  - Resolution $[32^3, 128^3] = [32 \text{ kB}, 2 \text{ MB}]$

- World SDF generated by combining N brushes
  - Each brush has translation, rotation and uniform scale
  - Smooth add/cut operations (exponential min/max)
  - Layering system (operation ordering)
  - Runtime performance not dependent on brush count
Compute Shader Intro

- **SPMD** = single program, multiple data
  - My slides are written from perspective of one thread
  - Unless line starts with: “Group“

- Thread groups
  - Compute dispatches are split to thread groups
  - Sync barrier + groupshared memory (**GSM**)
World SDF Generation on GPU

1. Generate SDF brush grid
2. Generate dispatch coordinates and mip masks
3. Generate level 0 in $8 \times 8 \times 8$ tiles (sparse)
4. Generate mips (sparse)
Generate SDF Brush Grid

64x64x32 dispatch. 4x4x4 groups

1. Sample a brush volume at tile center \( T \)
   1. Cull if SDF > grid tile bounds + 4 voxels
   2. Accepted? \( \rightarrow \) atomic add + store to GSM

2. Loop through brushes in GSM
   1. Sample \( \text{brushGSM}[i] \) at cell center \( C \)
   2. Accepted? \( \rightarrow \) store to grid (linear)
   3. Local + global atomic for compaction
Generate Dispatch Coordinates

64x64x32 dispatch. 4x4x4 groups

1. Read a brush grid cell
2. If not empty:
   1. Atomic add (L+G) to get write index
   2. Write cell coordinate to buffer
Generate Mip Masks

4x Dispatch (mips). 4x4x4 groups

1. **Group**: Load 1 voxel wider grid L_{-1} neighborhood
   1. Downsample count!=0 mask and store to GSM
2. Dilate mask by 1 voxel (3x3x3 GSM nbhood)
3. Mask!=0 → Write grid cell coords (prev slide)
Generate Level 0 (sparse)

**Indirect** Dispatch, 8x8x8 groups

1. **Group**: Read grid cell coordinate ($SV_{GroupId}$)
2. Read a brush from grid and store to GSM
3. Loop through brushes in GSM
   1. Sample brushGSM[i]
   2. Do exp smooth min/max operation
4. Write voxel to WorldSDF level 0
Generate Mips (sparse)

4x **Indirect** Dispatch (mips). **8x8x8** groups

1. **Group**: Load 4 voxel wider $L_{-1}$ neighborhood
   1. 2x2x2 downsample (avg) and store as $12^3$ in GSM
   2. $+\mathbf{4}$ voxel band becomes $+\mathbf{2}$ voxel band

2. **Group**: Run 3 steps of eikonal eq in GSM
   1. Expands band: 2 voxels $\rightarrow$ 4 voxels

3. Store **8x8x8** center of the neighborhood
Eikonal Equation (Wikipedia)

$n$-D approximation on a Cartesian grid  [edit]

Assume that a gridpoint $x$ has value $U = U(x) \approx u(x)$. Repeating the same steps as in the $n = 2$ case we can use a first-order scheme to approximate the partial derivatives. Let $U_i$ be the minimum of the values of the neighbors in the $\pm e_i$ directions, where $e_i$ is a standard unit basis vector. The approximation is then

$$\sum_{i=1}^{n} \left( \frac{U - U_i}{h} \right)^2 = \frac{1}{f_i^2}.$$ 

Solving this quadratic equation for $U$ yields:

$$U = \frac{1}{n} \sum_{i=1}^{n} U_i + \frac{1}{n} \sqrt{\left( \sum_{i=1}^{n} U_i \right)^2 - n \left( \sum_{i=1}^{n} U_i^2 - \frac{h^2}{f_i^2} \right)}.$$ 

If the discriminant in the square root is negative, then a lower-dimensional update must be performed (i.e. one of the partial derivatives is 0).

If $n = 2$ then perform the one-dimensional update

$$U = \min_{i=1,\ldots,n} (U_i) + \frac{h}{f_i}.$$ 

If $n \geq 3$ then perform an $n - 1$ dimensional update using the values $\{U_1, \ldots, U_n\} \setminus \{U_i\}$ for every $i = 1, \ldots, n$ and choose the smallest.
Eikonal equation

Distance field generated by repeatedly applying eikonal equation on a grid. Double widget to show error (SDF) compared to analytical solution.

```
float eikonal1d(float x, float v, float g)
{
    return min(x, v) + g;
}

float eikonal2d(float x, float y, float v, float g)
{
    float dy = dy + dy;
    float d = dy + dy - 2.0 * (x * x + y * y) - g * g);
    return 0.5 * (dy + sqrt(d));
}

void minImage(out vec4 color, in vec2 coord)
{
    int frame = iFrame;
    vec2 pixel = fragCoord;
    vec2 uv = fragCoord.xy / iResolution.xy;
    float distAnalytical = distFunc(pixel);
    if (frame == 1)
    {
        dist = distAnalytical;
    }
```
World Modification

- GPU simulated clay shapes
  - Up to 16k particles each
  - Smooth cut for each particle \(\rightarrow\) world collision
  - Shapes can also stamp copies of themselves (add)

- Fluid erosion
  - Up to 64k fluid particles
  - Smooth cut for each particle \(\rightarrow\) world collision
World Modification, cont

- SDF has infinite range
  - Local modifications are very expensive…

- Our volume texture has limited range!
  - 8-bit multilevel SDF
  - **Mip 0:** +/-4 voxel band around modification
  - **Mip 1+:** Dilate, but size = 12.5%, 1.6%, 0.2%…
  - \(\rightarrow\)** Efficient local modifications!**
World Modification, cont

- Same world generation algorithm, except:
  - Build grid with modifications only
  - Sample previous volume data at start...

- Must output to temporary buffer on PC
  - DirectX 11.1 (Win7) doesn’t support typed UAV load
  - In-place update of R8_unorm data can’t be done!
  - Workaround: Indirect dispatch to copy 8x8x8 tiles
Future: Sparse Volume?

- Only ~10% of mip0 8x8x8 tiles used
- Software virtual texturing with 8x8x8 tiles
  - Low res 3d indirection texture + 3d tile atlas
- Indirection texture read perf hit?
  - Our sphere tracing steps are fetch bound
  - Indirect = nearest (full rate) + trilinear (½ rate)
  - Measured cost = 13% slower
Ray-Tracing Distance Fields

- SDF(P) = distance to the closest surface at P
  - Radius of sphere at P (filled with empty space)
- Sphere tracing algorithm
  1. \( D = \text{SDF}(P) \)
  2. \( P += \text{ray} \times D \)
  3. \( D < \text{epsilon} \) → BREAK
Multilevel Volume Texture Tracing

Loop

\[
D = \text{volume.SampleLevel}(\text{origin} + \text{ray}\times t, \text{mip})
\]
\[
t += \text{worldDistance}(D, \text{mip})
\]
\[
D == 1.0 \rightarrow \text{mip} += 2
\]
\[
\text{IF } D \leq 0.25 \rightarrow \text{mip} -= 2; \ D -= \text{halfVoxel}
\]
\[
D < \text{pixelConeWidth} \times t \rightarrow \text{BREAK}
\]

- Break if surface is inside pixel inner bounding cone
  - Perfect LOD!
Last Step

- Sphere trace takes infinite steps to converge
- Assume we hit a planar surface
  - Trilinear filter = piecewise linear surface
- Geometric series
  - Use last 2 samples
  - Step = $D/(1-(D-D_{-1}))$
SDF Sweeps

- SDF can be swept by any bounded shape
  - Point sweep (ray): step by D
  - Sphere sweep: step by D – radius
- SDF cone trace (spherical cap)
  - Analytic solution exists
  - Only one extra instruction in shader!
Cone-Tracing Analytic Solution

**Pre-calculate (CPU):**
\[ C = \sqrt{\text{aperture}^2 + 1} \]
\[ A = \frac{C}{(C - \text{aperture})} \]

**In shader:**
\[ t = (t + D) \times A \]
Coarse Cone-Trace Pre-Pass

Cone-trace = Push growing “spheres” as far as they can go.

8x8 pixel (outer) bounding cones

Spawn pixel rays at end of the spherical cone
Future: Improving the "Edge Case"

Trace until cone cap is fully inside a surface. Store multiple in/out pairs for fast empty space skipping.
Ray Tracing Results

- Cone trace skips large areas of empty space
  - Huge step length reduction
  - Volume sampling more cache local
- Mip maps improve cache locality
  - $\log_8$ scaling of data: 100%, 12.5%, 1.6%, 0.2%...
- Measurement (1080p render)
  - 8 MB data accessed (512 MB). 99.85% cache hit rate
Failed Techniques: Overstepping

- **Idea:** Take longer steps
  - $\text{dist}(P_1, P_2) \leq SDF(P_1) + SDF(P_2)$
  - **Fail** $\rightarrow$ Rollback to previous sample

- **Problems:**
  - Reduces sampling cache locality (random rollback)
  - $SDF(P)$ more noisy with our mipmapped approach
  - Bloats VGPR count and adds ALU
Failed Techniques: Load Balancing

- Loop continues until all threads in wave exit
  - Some rays need significantly more steps than others
- **Idea:** Use wave ballot to exit loop early
  - 50% rays finished → fill finished threads with new rays
- **Problems:**
  - Ray setup code runs for unfinished rays (<50%)
  - Volume texture sampling is less cache local
- Coarse cone-trace is simpler and does the job better
Ambient Occlusion

- Cast cone at surface normal direction
  - Add random variation + temporal accumulate
- AO rays use low SDF mip
  - Better GPU cache locality and less bandwidth
  - Soft long distance AO
- We also use UE4 SSAO
  - Small scale (near) ambient occlusion
Soft Shadow Sphere-Tracing

- Soft penumbra widening shadows
- Approximate max cone coverage by stepping SDF along light ray
- Demoscene cone coverage approximation [1]:
  \[
  c = \min(c, \text{light}_\text{size} \times \text{SDF}(P) / \text{time})
  \]

Soft Shadow: Our Improvements

- Triangulate closest distance
  - Demoscene = single sample (min)
  - Triangulate cur & prev samples
  - \(\rightarrow\) Less banding

- Jitter shadow rays
  - UE4 temporal accumulation
  - Hides remaining banding artifacts
  - Wider inner penumbra
# Ray-Tracing Timings

<table>
<thead>
<tr>
<th></th>
<th>Xbox One (base) @ 720p</th>
<th>AMD Vega @ 4K</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cone-trace pre-pass</td>
<td>0.2 ms</td>
<td>0.2 ms</td>
</tr>
<tr>
<td>Primary &amp; AO rays</td>
<td>1.5 ms</td>
<td>1.6 ms</td>
</tr>
<tr>
<td>Shadow rays</td>
<td>1.7 ms</td>
<td>1.9 ms</td>
</tr>
<tr>
<td>Material &amp; g-buffer</td>
<td>0.8 ms</td>
<td>1.0 ms</td>
</tr>
</tbody>
</table>

60 fps target on all consoles
Clay Simulation

- Position based dynamics (PBD) on GPU
- SDF based clay shapes
  - $64^3$ SDF converted to point cloud for physics & render
  - Up to 16384 particles per clay shape (surface)
- Collisions to world SDF and between shapes
  - $O(1)$ particle<-SDF collision detection!
  - Plastic deformation
SDF \rightarrow \text{Mesh Conversion}

- Two pass approach
  - Multiple triangles refer to the same particle
  - Need to generate the particles first

- Output
  - Linear array of particles (surface) for PBD simulator
  - Index buffer for triangle rendering

- All meshes drawn with a single indirect draw call
SDF \rightarrow \text{Mesh Conversion (Particles)}

64 \times 64 \times 64 \text{ dispatch. } 4 \times 4 \times 4 \text{ groups }

1. **Group:** Load $6^3$ SDF neighborhood to GSM

2. Read $2^3$ GSM nbhood, if found in/out edge →
   1. Move P to surface (gradient descent)
   2. Allocate particle id (L+G atomic)
   3. Write P to array[id]
   4. Write particle id to $64^3$ grid
SDF $\rightarrow$ Mesh Conversion (Triangles)

$64 \times 64 \times 64$ dispatch. $4 \times 4 \times 4$ groups

1. **Group**: Load $6^3$ SDF neighborhood to GSM
2. Read $2^3$ GSM nbhood, if found XYZ edge →
   1. Allocate 2x triangle per XYZ edge (L+G atomic)
   2. Read 3x particle ids from $64^3$ id grid
   3. Write triangle to index buffer (3x particle id)
Shape Morphing

- Linearly interpolate between two SDFs
- Run SDF → mesh generation every frame
Ray-Traced SDF Meshes?

- Render SDF mesh bounding box to g-buffer
  - Vertex shader outputs local ray start point and direction
  - Pixel shader sphere-traces mesh volume
- Ray miss $\rightarrow$ discard pixel
- Use conservative depth (SV_Depth_LessEqual)
  - Up to 6x faster than SV_Depth when high overdraw
- Didn’t use this as our deform is particle based!
Shape Matching Solver

● 60 Hz fixed step length (16.6 ms)
  ● One constraint solve per physics tick

● Reductions:
  ● Group per body (1024): 16x loop load + reduce in GSM
  ● Reduce 3x3 covariance matrix
  ● Solve 3x3 SVD/PD \(\rightarrow\) rotation matrix

● Ported SVD/PD solver CUDA \(\rightarrow\) HLSL (MIT license)
Failed Techniques: Verlet Integration

- 1\textsuperscript{st} order technique
- Only position data
- **Problem:**
  - Linear estimate of $P_{+1}$
  - Projection damps rotation
- **Solution:**
  - Use 2\textsuperscript{nd} order integrator (**BDF2**)
Failed Techniques: Gauss-Seidel

- Graph colorization
  - Split constraints to 32 passes (independent)
- Constraint passes solved in GSM
  - No memory traffic between passes
- Performance and stability very good!
- **Problem**: GSM limited to ~2000 particles/shape
Failed Techniques: Jakobi

- Sum constraint projections, divide by joint count
  - Parallelizes perfectly
  - No limits for constraints
- Successive over relaxation (SOR) = 2x speed up
- **Problem**: Required 4x more sub-steps vs GS
  - Converges too slowly…
Fluid Simulation

- Smoothed Particle Hydrodynamics (SPH)
  - Clay fluid = highly viscose + smooth surface
  - 64k fluid particles (25cm radius)
- Fluid rendering
  - Generate fluid SDF every frame
  - Resolution = $256^3 + 1$ mip
  - Ray-traced (prim, AO, shadow)
Recommended Physics Papers

Collections of GPU simulation papers:

- http://matthias-mueller-fischer.ch
- http://mmacklin.com
Async Compute

● Split frame to 3 async segments
  ● Overlap UE4 g-buffer and shadow cascades
  ● Overlap UE4 velocity render and depth decompress
  ● Overlap UE4 lighting and post processing

● Work submitted immediately
  ● Compute queue waits for a fence to start (x3)
  ● Main queue waits for fence to continue (x3)
FPS increase = 19%+
Integration to UE4 renderer

- **G-buffer combine**
  - Full screen PS to combine ray-traced data
  - Samples material map (custom gather4 filter)
  - Writes to UE4 g-buffer + depth buffer (SV_Depth)

- **Shadow mask combine**
  - Full screen PS to sphere trace shadows
  - Writes to UE4 shadow mask buffer (with alpha blend)
UE4 RHI Customizations

- Set render target(s) without implicit sync
  - Can overlap depth/color decompress
  - Can overlap draws to multiple RTs *(image)*
- Clear RT/buffer without implicit sync
- Missing async compute features
  - Buffer/texture copy and clear
- Compute shader index buffer write
Thanks!

- UE4 Rendering Team
- Rys Sommefeldt (AMD)
- Lou Kramer (AMD)
- Adam Miles (Microsoft ATG)

More questions? We have ID@Xbox station in South Hall Lobby Bar (Thu/Fri)
Bonus Slides

● UE4 Build Process
● UE4 Merging
● UE4 Customizations
● UE4 Optimizations and Fixes
● Implementation Notes
Built on Top of Unreal Engine 4

- UE4 = huge code base + lots of shaders
  - Needs fast development hardware
- 16-core AMD Threadripper workstations
  - UE4 build system scales well to 32 threads
  - Around 3x faster build time vs 4 GHz i7 quad
- Large SSDs for checkouts
  - Gigabytes of symbol and .obj files
Unreal Engine 4 Merging

- Started with UE 4.8. Now UE 4.18
- Merged most major UE4 versions
- Created our own 3-way directory merge tool
  - UE4 console source code comes as zip package
- Will merge UE 4.19 soon
  - New features = temporal upscaler + dynamic resolution
Unreal Engine 4 Customizations

- Early decision: Fully separate our tech
  - Our own UE4 module
  - C-header with function entry points
  - 1-line modifications around UE4 code to call our module

- Separation not possible for all cases
  - UE4 RHI + low level changes (GPGPU features)
  - UE4 WorldCollision changes (SDF collision)
UE4 RHI Customizations (Extra)

- GPU->CPU buffer readback
  - UE4 only supports 2d texture readback without stall
  - Other readback APIs stall the whole GPU
- Buffer can have both raw and typed view
  - Wide raw writes = fill narrow typed buffers efficiently
UE4 optimizations

• Allow overlap of indirect dispatches/draws
• Allow overlap of clears and copy operations
• Allow overlap of draws to different RTs
• Reduced GPU cache flushes and stalls (image)
• Optimized staging buffers
• Fast clear improvements
UE4 optimizations

- Optimized barriers and fences
- Optimized texture array sub-resource barriers
- Better GPU tile modes for 3D textures
- Improved partial 2D/3D texture updates
- 5x faster histogram + eye adaptation shaders
- 4x faster offline CPU SDF generator (cooking)
Implementation Notes

- Physics data stored in one big raw buffer
  - Wide Load4/Store4 instructions (16 byte), bit packed:
  - Particle positions: 16 bit norm
  - Particle velocities: fp16
  - Bitfield for particle flags (alive, collided, etc)
- Benchmark tool: https://github.com/sebbbi/perftest

- Groupshared mem was a big performance win
  - SDF generation, grid generation, physics
  - Use when doing repeated loads of same data
Implementation Notes (2)

- Scalar loads were a big performance win on AMD
  - **Use case:** Constant index raw buffer loads
  - **Use case:** SV_GroupID based raw buffer loads
  - → Load stored to SGPR → Better occupancy