High Zombie Throughput in Modern Graphics

Anton Krupkin
CTO and Founder, Saber Interactive

Denis Sladkov
Lead Graphics Programmer, Saber Interactive
World War Z

- 4-player cooperative 3rd-person shooter
- Massive zombie crowds
- Powered by Saber Engine
- XBox One / Playstation 4
  - 30 fps
  - Up to 4k dynamic resolution
- PC
  - DirectX 11 and Vulkan
World War Z
Agenda

● Saber Engine render pipeline
  ○ High-level GPU workflow
  ○ Shading overview
  ○ Vulkan optimizations
Agenda

● Saber Engine render pipeline
  ○ High-level GPU workflow
  ○ Shading overview
  ○ Vulkan optimizations

● Zombies rendering
  ○ Close-up and mid-range characters
  ○ Far-plane swarm
  ○ Decals and gibbing
Saber Engine render pipeline

- GPU-driven visibility system
- Full depth prepass
- Forward+ PBR shading
- Baked GI: RNM + dominant direction
- 2 frames of latency
- Multithreaded command buffer recording
ZPass : depth + vertex normals
Shadowmaps: PCF only
Shadowmaps: filtered
Lighting: indirect
Lighting: indirect + direct
Occlusion: pre-baked with GI (stat scene)
Occlusion: SSAO
Reflections: off
Reflections: Cubemaps + SSR
Volumetrics: shadow light tracing [Glatzel14]
GPU: Forward+ Shading

- Raster
  - Compute
    - Shadowmaps
      - SSAO
      - Capsule AO
      - SSR
      - Volumetrics
      - Light/Cubemap
      - Froxel Bitmask
    - Forward+ Shading
      - Transparent Particles/FX
      - Motion Blur
      - TAA
      - Tonemap
    - Upscale
    - GUI
    - Present
    - ZPass (FRAME 0)
    - ZPass (FRAME 1)

CPU readback
GPU: Visibility Test

- ZPass (FRAME 0)
  - Raster
  - Compute
  - Shadowmaps
    - SSAO
    - Capsule AO
  - SSR
  - Volumetrics
  - Light/Cubemap
    - Froxel Bitmask
  - Visibility Test

- Forward+Shading
  - Transparent Particles/FX
  - Motion Blur
    - TAA
    - Tonemap
  - Upscale
  - GUI
  - Present

- ZPass (FRAME 1)

CPU readback
GPU Visibility system

- Based on “Practical, Dynamic Visibility for Games” [Hill11]
- Simple and efficient, no need for PVS/portals etc.
- Main camera: HZB occlusion culling with handmade occluders
- PSSM splits: frustum culling only
- 0.7ms (XBox One) GPU budget for scene with 100k+ objects
- Requires 1 frame of latency for CPU readback
GPU Visibility
GPU Visibility: occluders
GPU Visibility: culling efficiency
GPU Visibility: culling efficiency
Vulkan vs D3D11

- Up to 10% less GPU time
  - Wave intrinsics (*GL_KHR_shader_subgroup*)
    - Forward+ lighting / reflection loops scalarization in shading pass
  - Async compute
  - Half-precision math
- No helper driver thread: 40% less total rendering CPU load
- 20% reduction of CPU critical path due to MT command buffers
- Render target memory aliasing
  - 30% less memory
  - Dynamic resolution
Zombies rendering
Zombies rendering

- Over 5k visible zombies per-frame
- Most are non-interactive background swarm
- 300+ foreground “real” ingame entities/characters
- Only 50 have fully developed zombie brains
- Instancing was used to reduce draw call pressure
- Flexible per-instance visual customization system
  - Inspired by “Shading a bigger, better sequel” [Grimes10]
Customization: meshes

● 3 base archetypes / skeletons
  ○ Male / Female / ‘Big’ male
  ○ ~50 bones per skeleton

● 4 mesh regions per archetype
  ○ Legs / Torso / Head / Hair
  ○ 2-5k tris per region

● 4-8 mesh variations for each region

● 70+ unique mesh combinations total
  ○ Plus ~10 additional unique zombie models (chemical, screamer, etc...)
Customization: male archetype meshes
Customization: color and stain masks

● For each mesh region (shirts / jeans / shoes / hair)
  ○ Color masking texture (ARGB8 texture, low-res)
  ○ Stain masking texture (ARGB8 texture, low-res)

● Shared stain texture set (bloodstains / snow / dust)
  ○ Albedo / normals / roughness textures
  ○ All textures are tiled
  ○ Stored as texture array

● Per-instance constants
  ○ Color: Channel mask + solid color
  ○ Stain: Channel mask + texture array index
Customization: color and stain masks

TORSO MASK:

LEGS MASK:
Customization: hundreds distinct looks
Zombie rendering: instancing

- Get individual zombies from visibility system
- Gather identical meshes into instancing buckets
  - Same mesh variation for one region going into one bucket
  - Different LOD models going to separate buckets
- Process accumulated buckets into render queue
  - Sort bucket by visibility mask (each active camera has it’s own bit)
  - Gather meshes with the same visibility mask into batches (30 meshes max)
  - For each batch fill continuous constant buffer with per-instance data
  - Generate 1 draw call per batch for every pass/camera (Z, SM, shading)
Zombie rendering: sample workload

- **LOD 0 (8 meters):** 10 zombies, ~130k tris, 500 bones, 26 dp, 140k cb
  - 6 unique heads, 8 torsos, 6 legs, 6 hairs = 26 draw prims calls (vs 40)
  - 50 bones x 2 frames + customization = 100kb of per-instance consts
  - Per-draw call material constants: 40kb

- **LOD 1 (14 m):** 30 zombies, ~180k tris, 1500 bones, 38 dp, 520k cb
- **LOD 2 (25 m):** 70 zombies, ~175k tris, 2800 bones, 46 dp, 1120k cb
- **LOD 3 (35 m):** 100 zombies, ~115k tris, 2600 bones, 61 dp, 1250k cb
- **LOD 4 (>35m):** 100 zombies, ~25k tris, 2600 bones, 52 dp, 1130k cb

- **Total:** 223 draw calls (vs 1240), 625k tris, ~4mb of const buffer data
- **2.5ms x 6 threads CPU to generate ~1000 draw calls (Z, SM, Shading)**
Zombie rendering: background swarm
Zombie rendering: background swarm

- Over 5k zombies
  - Non-interactive: essentially just a GPU-driven animation
  - 8 pre-baked variations total: 2 mesh types x 4 unique appearances
  - ~400 triangles per mesh
- Inspired by “The Technical Art of Uncharted 4” [Maximov16]
- Utilize existing grass rendering solution
- Add texture-baked vertex animation
- Move alongside pre-modeled tracks
Swarm: grass rendering solution

- Zombies are authored as grass blades
- Blades are distributed with ‘seed brush’ over level geometry
- Same blades spatially merged into ‘containers’
- Each container has
  - Per-blade instancing data
    - Position + orientation
    - Indirect lighting (sampled from underlying LM)
    - Seed for procedural animation and track selection
  - OBB for visibility culling / LOD calculations
  - Single draw call to render all blades
Swarm: baked vertex animation

- Just a single ‘running’ animation for each of 2 zombie types
- Stored as per-frame per-vertex offset + normal
- Baked in two 32-bit textures:
  - Offsets from default model - quantized to 8 bit using model’s AABB
  - Object-space normals stored as is
- VS: Sampled with UV = (vertex_id, frame_number)
Swarm: movement tracks

- Designers lay tracks for a running swarm
- Each track is authored as a spline
- Each track converted to an array of 16 bit fixed point positions
- Whole set of tracks stored as single 2D INT16 texture
- VS: Sampled with UV = ( elapsed_time, track_index )
Gibbing & decals
Gibbing & decals

- Hide triangles in wounded regions using vertex masks
- Unhide parts of pre-modeled gore meshes
- Add decals for better mesh - gore mesh stitching
- Based on “Rendering wounds in Left 4 Dead 2” [Vlachos10]
Gibbing: region vertex masking

- Define mesh regions to be hidden during gibbing
  - Use original mesh tessellation
- Store per-vertex 32-bit mask
  - 1 bit per region, 32 regions max
- Split the edges on region’s border
- Define per-instance 32-bit mask
  - Zero bits for gibbed regions
- VS: hide masked triangles
  - $M_{\text{vert}} \& M_{\text{inst}} = 0 \rightarrow$ zero-area tri
Gib meshes
Decal system

- Transform damage origin to model bind pose (inverse skinning) and store it
- Pass bind pose vertices from VS to PS
- PS: loop and apply decals before lighting
  - Planar projection with angle fade-factor
  - Simple blend mode
  - Albedo + normal + roughness / metalness
  - 16 decals max
References

[Vlachos10] “Rendering wounds in Left 4 Dead 2”, Alex Vlachos, GDC 2010
[Iwanicki13] “Lighting Technology of The Last of Us”, Michal Iwanicki, Siggraph 2013
Thanks

AMD
- Adam Sawicki

Saber
- Sergey Demidov
- Kirill Arefyev
- Mikhail Korovkin
- Nick Petrov
- Max Gridnev
- Alexander Smetkin

- Jordan Logan
- Alexander Skolunov
- Dmitry Zaborovsky
- Ivan Shostak
- Ivan Popov
- Timur Popov
- Anton Vasilev
Questions?

krupkin@saber3d.com
sladkoff@saber3d.com
Bonus slides
GPU Visibility system: input

- Immutable UAV buffer for stat scene
  - 2-level world space OBB hierarchy

- Semi-dynamic UAV buffer for moveable objects
  - Allocated on-demand via free list
  - Occasional per-element OBB update
  - Used for any type of objects: characters/lights/volumetrics etc.

- List of cameras
  - Frustum + set of control / filter flags
  - Includes main camera and any number of dependent cameras (SM)
GPU Visibility system : HZB

● Render occluders to fixed resolution depth (512x256)
  ○ About 10k triangles total - really low res occluders
  ○ Several dozen draw calls - occluders merged and frustum-culled on CPU

● Build HZB - conservative depth mip hierarchy down to 2x2

● OBB occlusion test
  ○ Calculate bbox screen-space extents
  ○ Choose appropriate HZB level (2x2 pixels bbox coverage)
  ○ Cull by OBB min depth
  ○ Subpixel culling
GPU Visibility system: tests

- Camera specific set of tests
- Frustum vs OBB (all cameras)
- Occlusion (main camera only)
- Distance / screen-space area (SM cameras)
- Object state vs camera state (LoDs, reflection camera)
GPU Visibility system: output

- Per-object cam mask & distance list
- Hierarchical merge stage (2-level hierarchy for stat scene)
- Per-camera, per-object type append buffer with object IDs
- CPU readback
- Object ID list used as direct input for CPU render jobs
  - ZPass / Shadowmap
  - Volumetrics / Dynamic lights / Reflection probes
GPU Visibility: drawbacks

- 1 frame of latency for CPU readback
- Manual occluder authoring
- 2 frames of latency for SM tests in case of light culling test