AGENDA

RAINBOW RENDERER OVERVIEW
MATERIAL BASED DRAW CALL SYSTEM
CHECKERBOARD RENDERING
AGENDA

RAINBOW RENDERER OVERVIEW

MATERIAL-BASED DRAW CALL SYSTEM

CHECKERBOARD RENDERING
• Rebirth of a loved franchise that is precious to a lot of hard-core gamers

• Gameplay driven

• Destruction as a core gameplay mechanic
  • Destruction must be consistent between platforms

• First Rainbow Six shipping with the engine
  • Lots of legacy code from previous prototypes
SIEGE TECH MISSION

- Targeting **60FPS**:
  - GPU: 14ms average on non combat situations
  - CPU: Max 38ms linear time on CPU (consoles)
- Provide scalable destruction

- Ship at a higher resolution than 720p on all consoles
- Commit to 4K on PC at a decent framerate

- Provide a strong PC version on a console oriented production
  - Fitting on 1GB of RAM becomes a challenge with current gen
SIEGE IS A LIVE GAME

• Graphics features can be continually iterated on

• Test new tech to improve look or comfort
  • Auto-exposure for players

• Be careful not to break things!
Hierarchical view of a GPU frame

- Average 5ms spent on geometry rendering
  - Heavy use of culling!
  - Shadow caching!

- Average 5ms spent on lighting (SSR included)
  - Checkerboard rendering helps!
  - SSAO & SSR ray trace done in async

- Average 4ms spent on post processing/other full screen processing
**SIEGE FRAME**

- Hierarchical view of a CPU critical path
- 10ms avg on the critical path
  - All passes and tasks able to fork and join to minimize critical path
  - Shadow caching!

- Max 4ms linear spent on opaque pass
- Material based draw call system!
First person rendering → 400 best occluders to depth buffer → Generate Hi-Z → Opaque culling & rendering
SHADOW RENDERING

• All shadows are cache based
  • Use cached Hi-Z for culling

• Sunlight shadow done in full resolution
  • Separate pass to relieve lighting resolve VGPR pressure
  • Uses Hi-Z representation of the cached shadow map to reduce the work per pixel

• Local lights are resolved in a quarter resolution
  • Resolved results stored in a texture array
  • Lower VGPR usage on light accumulation
  • Bilateral upscale
**SHADOW RENDERING – SUN / MOON**

- Shadow map containing all static objects built on load

6Kx6K 16bit

512x512 ESM

Hi-Z
**SHADOW RENDERING — SUN / MOON**

- Ability to scale shadow cost by mixing cascades with static map
  - Static Hi-Z shadow map always used for dynamic object culling

- On Xbox One:
  - 1st cascades are fully dynamic (not enough resolution with 6K)
  - 2nd and 3rd cascades renders dynamic objects only and blend with the static shadow map
  - 4th cascade is substituted by the static shadow map
SHADOW RENDERING – LOCAL PROJECTORS

• We handle a maximum of 8 visible shadowed local lights
LIGHTING

• Uses a clustered structure on the frustum:
  • 32x32 pixels based tile
  • Z exponential distribution

• Hierarchical culling of light volume to fill the structure

• Local cubemaps regarded as lights

• Shadows, cubemaps and gobos reside in textures arrays
  • Deferred uses pre-resolved shadow texture array
  • Forward uses shadows depth buffer array
AGENDA

RAINBOW RENDERER OVERVIEW

MATERIAL BASED DRAW CALL SYSTEM

CHECKERBOARD RENDERING
RAINBOW SIX DESTRUCTION

ART DIRECTION
• When destruction happens you need to feel that something big went on!

FLOORS & WALLS
• Procedurally generated unique geometry
• Poking holes degrades occlusion efficiency

DESTRUCTIBLE PROPS & DEBRIS
• Generally smaller meshes but in great numbers
• Can be instanced or unique
UNIQUE GEOMETRY
A PROP AABB THIS BIG WILL FAIL OCCLUSION TEST
RAINFOREST DESTRUCTION

• Early prototypes were largely graphic bound (CPU and GPU) on average

• PC DX11 deferred contexts aren’t that great at scaling

• Material based draw call system
  • Materials define destruction properties
  • Debris share material
  • See [Haar&Altonen15]

• In need of granularity in culling to keep up with destruction
UNIFIED BUFFERS

A lot of resources in Rainbow Six reside in an unified buffer of some sort:
- Unified Vertex Buffer
- Unified Index Buffer
- Unified Constant Buffer
- ...

Structured buffers built on top of raw buffers with auto generated code:
- Using C++ data descriptors for GPU unified data
- Meta data passed on to specify access pattern
UNIFIED BUFFERS - CONSTANT

// Cpp Code
void EntityConstantBufferBridge::CreateDescriptor(ConstantBufferDescriptor* descriptor)
{
    popBeginMapShaderParametersAndSetUsage(EntityConstantBufferBridge, descriptor, Entity);
    popMapShaderParameter("g_NodeWorld", m_WorldMatrix);
    popMapShaderParameter("g_InvScaleSqr", m_InvScaleSqr);
}

// --

// HLSL GeneratedCode
ByteAddressBuffer g_UnifiedConstantBuffer;
void LoadEKInstanceProvider(uint index)
{
    uint offset = Mad_U24(index, EKINSTANCENONE provider STRIDE, EKINSTANCENONE provider GLOBAL BUFFER OFFSET);
    g_NodeWorld = ToFLOAT4x4(
        g_UnifiedConstantBuffer.Load4(offset + 0x0),
        g_UnifiedConstantBuffer.Load4(offset + 0x10),
        g_UnifiedConstantBuffer.Load4(offset + 0x20),
        g_UnifiedConstantBuffer.Load4(offset + 0x30)
    );
    g_InvScaleSqr = ToFLOAT4(g_UnifiedConstantBuffer.Load4(offset + 0x40));
}

// --
UNIFIED BUFFERS - BENEFITS

• Complete control over data layout
  • We can easily experiment with different data type accesses (AOS, SOA, Structure of u32 Arrays...)
  • Custom packing and support for new data types

• High level API supports broadcasting values

• Code auto-generation allows us to migrate to new access patterns easily
MATERIAL BASED DRAW CALLS

• Geometry and constants are unified

• A draw call is then defined by:
  • Shaders
  • Non-Unified Resources (Textures, etc...)
  • Render States (Sampler States, Raster States)

• Elements that share the above are batched together

• Passes that don’t use a subset of the resources and states are further batched together
GATHERING DRAW CALLS

- On initialization, each submesh instance is mapped to 3 batches: Normal, Shadow and Visibility
- The batch types used to mask non necessary data
- Each batch will correspond to a MultiDrawIndexedIndirect command
GATHERING DRAW CALLS

- Each submesh instance has a globally unique index:
  - Index used to fetch all data
  - Multiple indirection needed
GATHERING DRAW CALLS

• For each pass gather the submesh instance index into a dynamic buffer:
  • Each pass maps to one batch type exclusively
  • Buffer filled in multithreaded jobs (1.5ms linear)

• Extra data to perform culling is added:
  • MultiDrawIndexedIndirect entry
  • New index buffer offset
  • Additional culling flags
PERFORMING CULLING

We define multiple types of culling:

- **Level 1**: Submesh instance culling
- **Level 2**: Submesh chunk culling
- **Level 3**: Submesh triangle culling
PERFORMING CULLING

LEVEL 1 CULLING
- Screen Space Size Culling
- Distance Culling
- Frustum Culling
- Occlusion Culling

LEVEL 2 CULLING
- Screen Space Size Culling
- Frustum Culling
- Orientation Culling
- Occlusion Culling

LEVEL 3 CULLING
- Triangle Normal Culling
PERFORMING DRAW CALLS

PASS BUFFER
- SUBMESH INSTANCE 1
- DRAW BUFFER OFFSET
- INDEX BUFFER OFFSET
- CULLING FLAGS

- SUBMESH INSTANCE 2
- DRAW BUFFER OFFSET
- INDEX BUFFER OFFSET
- CULLING FLAGS

CULLING

MULTIDRAW BUFFER
- BATCH 1
  - MULTIDRAW PARAMETERS
  - MULTIDRAW PARAMETERS
  - ...

- BATCH 2
  - MULTIDRAW PARAMETERS
  - MULTIDRAW PARAMETERS
  - ...

...
PERFORMING A DRAW CALL

- Dynamic Index Buffer
- Unified Vertex Buffer
- Submesh Instance Index Buffer
- Unified Constant Buffer
- Submesh Instance Index
- Mesh Descriptor
- Vertex Data

**Draw Call Entry Parameters**

- Index Count (N)
- Instance Count (1..N)
- Start Index
- Base Vertex (0)
- Start Instance
PERFORMING A DRAW CALL

Culling compute shader writes out instance indices in a Per Instance Buffer.

```c
struct VertexShaderInput {
    uint PrimitiveInstanceId : PerInstanceInfo;
    uint VertexID : SV_VertexID;
};

void GetVertexFromUnifiedBuffer(uint clusterFirstVertexOffset, uint vertexIdx, out UnpackedVertexFormat vertex) {
    uint offset = clusterFirstVertexOffset + vertexIdx * VERTEX_FORMAT_STRIDE;
    vertex.Position = ToFLOAT3(g_VertexBuffer.Load3(offset + 0));
    vertex.Normal = ToUBYTE4(g_VertexBuffer.Load (offset + 12));
    vertex.TexCoord0 = ToFLOAT16_2(g_VertexBuffer.Load (offset + 16));
    vertex.VertexID = vertexIdx;
}
```
PERFORMING A **DRAW CALL**

ReadFirstLane is used in the pixel shader when loading UCB values with UniformsOffsets to be able to use the GCN scalar unit & registers.
RAINBOW SIX DESTRUCTION

+5 DRAWCALLS
### RESULTS

<table>
<thead>
<tr>
<th>NUMBER OF UNBATCHED DC (TOTAL)</th>
<th>NUMBER OF BATCHED DCS (VIS + GBUFFER + DECALS)</th>
<th>NUMBER OF BATCHED DCS (SHADOWS)</th>
<th>CULLING EFFICIENCY</th>
</tr>
</thead>
<tbody>
<tr>
<td>10537</td>
<td>412</td>
<td>64</td>
<td>73%</td>
</tr>
</tbody>
</table>
FUTURE WORK

• Pushing empty draw calls has a cost
  • We try to hide it on consoles using async jobs
  • Specifying the number of draw calls on the GPU would be the next step

• Using bindless resources to further batch draw calls

• Moving most of the scene graph traversal to the GPU
  • LoD selection logic
AGENDA

RAINBOW RENDERER OVERVIEW
MATERIAL BASED DRAW CALL SYSTEM
CHECKERBOARD RENDERING
60 FPS MADE EASY

• We wanted 60FPS early in production
  • First playable was running at around 50 on consoles
  • 60 FPS average was hit a couple weeks after!

• Killzone approach seemed like a good idea to start with (see [Valiant14])
  • Keeping nearly the same budget per pixel as a 30FPS game for screen pixels rendering
  • EQAA based, we wanted it on PC too (low end and 4K support)

• Big “quick” win without having a major quality impact
  • Silently enabled to see if people noticed
TEMPORAL INTERLACED RENDERING

- To target 1920x1080:
  - We render geometry and lighting to 960x1080 render target

- 3D velocity vector per rendered pixel
  - R12G12B8 format

- Projection matrix is offset each frame

- Need to divide x gradient by 2 to have similar texture filtering

```c
float4 SampleTexture2D(Texture2D t, SamplerState s, float2 coord)
{
    #if !INTERLACED_RENDERING
        return t.Sample(s, coord);
    #else
        float2 dx = ddx(coord);
        float2 dy = ddy(coord);
        return t.SampleGrad(s, coord, dx/2, dy);
    #endif
}
```
TEMPORAL INTERLACED RENDERING

- Things not represented by motion on screen need to be dealt with
  - Tried to maintain lighting/shadow changes to handle them better
  - Color clamping (See [Karis14])

- Data tweaked so alternating effects take place over at least two frames
  - Police car flash lights, light flickering
  - Flickering oscillators modified to avoid single frame 0 to 1 transitions

- Aliasing on vertical lines
- Not that easy after all!
CHECKERBOARD RENDERING

• Base idea came about to solve aliasing issues

• Experimented on a series of images to first test quality

• For most images PSNR was better using a checkerboard pattern:
  • Visually the results were more pleasing too

• The idea of using MSAA 2X was bouncing around since the beginning
  • We made a push for it for E3 2015
LINE NEIGHBORS INTERPOLATION
CHECKERBOARD NEIGHBORS
INTERPOLATION
LINE NEIGHBORS INTERPOLATION
CHECKERBOARD NEIGHBORS
INTERPOLATION
LINE NEIGHBORS INTERPOLATION
CHECKERBOARD NEIGHBORS
INTERPOLATION
CHECKERBOARD RENDERING IMPLEMENTATION

- Rendering to a ¼ size (½ width by ½ height) resolution with MSAA 2X:
  - We end up with half the samples of the full resolution image
- D3D MSAA 2X standard pattern
  - 2 Color and Z samples
- Sample modifier or SV_SampleIndex input to enforce rendering all sample
- Each sample falls on the exact pixel center of full screen render target
Checkerboard Rendering Bonus

- Particle effects can be easily evaluated per pixel instead of per sample
- You can fit a lot more stuff in ESRAM!
- No need to fixup gradients in the shaders!
With LOD bias

Texture gradients are represented by red lines

CHECKERBOARD RENDERING IMPLEMENTATION

GRADIENT FIXUP
By offsetting the projection matrix again each frame we are able to alternate the pattern.

We don’t always have access on PC to change sample locations.
FILLING IN THE BLANKS

To reconstruct colors for unknown pixels P and Q, we sample:
- Current frame direct neighbors linear-Z
- Current frame direct neighbors color
- History color and Z

Even frames

Odd frames
HISTORY COLOR/Z

• One neighbor gets picked for motion velocity:
  • Closest one to the camera to preserve silhouette

• With motion velocity we sample the previous resolved color
  • That way we get to use filtering, but introduces accumulation errors!

• We clamp the re-projected color with A B E F for Q

• Using previous depth computed from motion we compute a confidence value
  • Used to blend back toward the unclamped value.
RESOLVED COLOR

• Having:
  • The history color
  • The interpolated color from direct neighbors

• A final color is computed using two additional weights:
  • Color coherency:
    • Minimum difference between A B E F for Q
  • Magnitude of velocity
COMPLETE FLOW

• Resolve quite complex
  • Lots of tweaks for our content!

• Costs 1.4ms

• 8 – 10ms net win
T-AA

- Integrates with the checkerboard rendering
  - Can be run on the same resolve shader

- Done on the sub-sample level, MSAA 4X style jitters on top of the checkerboard pattern

- Reprojected color weight uses similar logic

- Additional “Unteething” used to remove bad checkerboard patterns
**TEETH REMOVAL FILTER**

- Resolve can introduce noticeable saw tooth patterns in the image
  - We apply a filter to remove them

- The filter works on 5 horizontally or vertically adjacent pixels

- We setup a threshold \( d \) and binaries pixel each to 0 or 1 if they fall in the range of \([0, d]\) or \([1 - d, 1]\)

- We detect a 01010 or 10101 pattern
FUTURE DEVELOPMENTS

• Checkerboard technique was a good win for us
  • We are going to push more quality per pixel and build up on it
  • Implementation mostly by trial and error, we will move to a more scientific approach on the different confidence weights and values used
SPECIAL THANKS

- Alexandre Lahaise
- Chen Ka
- Michel Bouchard
- Lionel Berenguier
- Paul Vlasie
- Stephen Hill
- Stephen McAuley
- Ulrich Haar
Thank you!

- RAINBOW RENDERER OVERVIEW
- MATERIAL-BASED DRAW CALL SYSTEM
- CHECKERBOARD RENDERING
QUESTIONS?
REFERENCES

• Karis14: https://de45xmedrsdbp.cloudfront.net/Resources/files/TemporalAA_small-59732822.pdf
• Valient14: http://www.slideshare.net/guerrillagames/killzone-shadow-fall-gdc2014-valient-killzonegraphics
• Hill11: http://blog.selfshadow.com/publications/practical-visibility/
• Intro video by 20Powerproductions: https://www.youtube.com/watch?v=Rc0V98BzW3g
• Icons by FLATICON.COM
**GBUFFER LAYOUT**

4 Render Targets \((\text{RGB10A2} + 3 \times \text{RGBA8})\) + Depth sStencil \((D32 – S8)\)

<table>
<thead>
<tr>
<th>R</th>
<th>G</th>
<th>B</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>World Normal (RGB10)</td>
<td></td>
<td></td>
<td>GI Normal Bias (A2)</td>
</tr>
<tr>
<td>BaseColor (sRGB8)</td>
<td></td>
<td></td>
<td>Config (A8)</td>
</tr>
<tr>
<td>Metalness (R8)</td>
<td>Glossiness (G8)</td>
<td>Cavity (sB8)</td>
<td>Aliased Value (A8)</td>
</tr>
<tr>
<td>Velocity.xy (RGB8)</td>
<td></td>
<td></td>
<td>Velocity.z (A8)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CONFIG</th>
<th>ALIASED VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default</td>
<td>Self A0 (sA8)</td>
</tr>
<tr>
<td>Skin</td>
<td>Skin SSS Mask (sA8)</td>
</tr>
<tr>
<td>Translucent</td>
<td>Translucence (sA7) + Back Face (A1)</td>
</tr>
</tbody>
</table>
**GBUFFER RENDERING**

- We use inverted depth combined with a D32 float for better uniform depth precision distribution.

- For normals we experimented with BFN first
  - We moved to a R10G10B10A2 format to save VGPRs and ALU.

- Velocity vector is 3D and enjoys a higher precision on the X & Y axis to support our temporal reprojection rendering.

- GBuffer Layer 2’s alpha is aliased depending on the material type
  - Self-AO was not used since SSBC revealed itself sufficient most of the time.
  - We apply a higher SSAO factor on the first person character.
GBUFFER RENDERING
LIGHTING - GI

• GI is static and is based on a simplification of Assassin’s Creed Unity GI
  • Low resolution volume covering whole map:
    • Sky visibility SH
    • 1m to 2m per voxel

• High resolution volume covering the playable area:
  • Sky visibility SH
  • Bounce color SH
  • 25cm per voxel

Screenshot of low res Volume  
Screenshot of high res Volume
**LIGHTING – DIRECT**

- We generate a clustered structure on the frustum:
  - 32x32 pixels based tile
  - Z exponential distribution
  - Hierarchical culling of light volume to fill the structure

- Light cookies (gobos) are gathered in an array to be able to fetch them dynamically
  - Simply part of the light data as indices in an array
LIGHTING – SSR

• Done in ¼ resolution

• Uses face normal to give ray direction

• Temporal reprojection with light accumulation (ray-based, not depth based)

• Linear marching, steps gloss dependent
  • Jitter start ray position and direction
  • Temporal reprojection smooth the results

• Invalidate previous frame result on camera movement
LIGHTING — REFLECTION

- Local cubemaps
  - Parallax corrected
  - Regarded as lights, volume injected in clustered lighting structure
  - Reside in cubemap array for easy access

- Cubemaps applied during SSR application
  - Local cubemaps are SSR’s primary fallback
  - Global cubemap is secondary fallback

Screenshot showing cubemap volumes
LIGHTING – FORWARD

• Support same set of features as the deferred pass:
  • All shadows, cubemaps, cookies are in texture arrays

• VGPR consumption issues:
  • Scaling down on the quality of shadow filtering
  • Glass disables some lights types
  • Still lowest occupancy in our renderer

• Expensive particles use the ESM version of the shadow cache
SCHEDULING

• Graphic thread managing work queues and stealing work when necessary, work stolen gets executed on the immediate context when possible to minimize overhead.

• On PC no draw calls are recorded we let the material based draw call pipeline handle the scaling.

• On consoles graphics work has priority on Cluster 0 — Core 0, 1, 2 and we also maintain cluster locality when scheduling tasks.
  • Fork & join work can take a turn for the worst when hammering shared atomics. (add numbers)
SCHEDULING

• Rendering-specific scheduler on top of the engine scheduler:
  • Full control of graphic task behavior to fit in our budgets
  • Task dependencies code defined
    • Investing on visualisation tools would have been worth while

• First implementation used system fibers
  • Workers can steal a job with more priority instead of waiting
  • Fibers confusing to programmers
  • Some systems have trouble displaying them properly in the debugger

• We moved to a simpler model where yielding just executes a new job on the current context
GRAPHIC CPU PERFORMANCE ON RAINBOW

• Beside from initialization, zero tolerance global allocator usage during the frame
  • Heavy use of per worker thread local allocator
    • Resets when outermost job finishes
    • Helps on cache locality and more flexible
  • Heavy use of pooling
    • Dangling pointers becomes harder
    • Adding memory state values on builds to check validity

• Memory access patterns were 95% of the optimization works on the graphic side
  • Per thread gather lists are used to decrease inter thread communication
  • Atomics have an important cost if not used properly
SLI SUPPORT

• Driver tracking disabled on all resources

• Simple scoping interface for update of resources that need sync
  • One line addition to the code when necessary

• Update of a couple of large buffers was implemented by propagating the changes manually on each GPU
  • A lot more efficient than synching buffers
  • Update of unified constant buffers takes a couple of μs, copy brakes scaling