

Resistance: Fall of Man

Insomniac Games

Development

- Started on PC with a small prototype team concurrent with PS2 development
 - Good for prototyping shaders – little change
 - Good for prototyping lighting, tools and build process
 - Bad for code development, systems were not written with the Cell in mind
- Later on, small group handled porting code base to PS3 while others continued working on PC

Development

- Many concepts shared from PS2
 - Asset types named the same to keep artists happy
 - Structure of the engine is similar
 - Similar tool chain
- No runtime code shared from PS2
 - PS3 Resistance and PS2 Ratchet had little in common
 - Lots of ASM on PS2 so not much to use on PS3
 - Most PS2 systems were considered too simple for PS3

Programmer Tools

- Perforce
- ProDG
- SN DBS
- Visual Studio
- In house asset control system
- In house build server
- In house world tool

Art Tools

- Maya
- Mental Ray
- Microwave
- Photoshop
- Z Brush
- SpeedTree

Asset Types

- Kept the same asset names to keep the process familiar
 - U-Frags
 - Ties
 - Mobys
 - Shrubs
 - Foliage
 - Skies
 - Shaders

U-Frags

- Most basic - fastest to render
- Completely static geometry
- Vertices stored in world space as locally compressed fragments to keep the precision as high as possible – real positions computed in the vertex programs
- Non-instanceable
- Lightmapped or vertex baked
- Memory efficient
- PVS occlusion

Ties

- Conceptually the same as U-Frag but instanced - vertices defined in local space
- Completely static geometry
- Can be nested within each other in the tools, invisible to runtime
- Lightmapped or vertex baked
- Made up the majority of a typical scene
- Stored as two RSX streams, a master and a per instance stream with the lighting info
- PVS occlusion

Mobys

- Dynamic geometry, most expensive
- Can have physics, animations, etc.
- May be rigid (1 bone) or skinned (4 bones)
- Static lighting from pre-computed spatial lighting volumes
- Dynamic lighting from runtime lights
- Crude static LOD

Shrubs

- Very fast to render, used to fill scene
- Levels would have 10k-20k instances
- Very basic geometry
- Simple lighting
- Didn't cast shadows but could receive them
- Basic wind like animation
- Fade out in distance
- PVS occlusion

Foliage

- Used for leaves on trees
- Camera facing billboards
- Used SpeedTree for data generation but wrote our own custom renderer for runtime
- Could cast and receive shadows
- Basic wind like animation similar to shrubs
- PVS occlusion

Skies

- Multiple frequencies of cloud layers composited together
- Artist driven animation parameters to procedurally control cloud turbulence, drift, formation rates, etc.
- Simplified geometry could be rendered before or after cloud layers
- Bloom geometry layers
- Drawn after opaque geometry
- PVS occlusion

Shaders

- Base color map, Alpha/Bloom, Normal, Gloss, Incandescence, Parallax, and Detail
- Fine grain control exposed to artists – texture format, filtering modes, etc.
- Vertex programs abstracted the asset type
- All assets shared fragment programs - consistent lighting
- Runtime has specific shader programs to optimally handle the various combinations of shader features
- Used simple pre-processor #defines to add / exclude program features
- Reduced texture formats when channels are missing, for example DXT5 can be DXT1 when there is no alpha
- Normal maps are stored compressed as 2 channels in a DXT5 map
- LDR Cubemaps created from probes placed in maya

Shaders

- The alpha type is part of the shader
 - Opaque (updates z)
 - Blended (does not update z)
 - Additive (does not update z)
 - Cutout (uses alpha test)
- Destination alpha used for bloom
- Shader LOD removed expensive features
- Shaders use the hardware features of the texture samplers
 - Swizzle to get the inputs in to the place where the fragment program expects them
 - Use the sign extend and force zero/one feature

Build Process

- Each asset type has a stand alone builder
 - PC Command line tool, integrated with our asset control system
 - Single platform, assets are built directly in hardware format
- PS3 viewer that shows individual built assets
 - Good for debugging assets
 - Good for rough stats on a single asset
 - Viewers show all physics and collision info
 - Viewers allow artists to light and animate out of the context of a level

Build Process

- Engine Data Level Packer
 - A level is a group of assets of different types packed together by the packing tool
 - The level pack tool is what lays optimally in the final format, resolves duplicates, partitions memory, etc.
 - On load, very little copying is done, we just need to patch up pointers
 - RSX data is packed into two chunks, one that ends up in main memory and one in local memory

Build Process

■ Problems

- No backwards/forwards compatibility of data formats
- Slow asset builds
- Live data

Static Lighting

- Environments use simplified HL2 style light maps - directional luminance in the lightmaps and a shared chrominance per vertex
- Environments could also use vertex lighting, in which case there was no lightmaps and the luminance and chrominance are stored per vertex
- The lightmaps, UVs and vertex components for lighting are stored per instance
- All lighting computed through Mental Ray
- Interactive lightmap resolution and format tweaking

Static Lighting

- Mobys and moveable objects use a static environment lighting database that is pre-computed from artist placed lighting volumes
- Luminance is stored per sample along 6-axes with a shared chrominance
- Data is stored as a grid and interpolated
- Code could query a single point or a cube and locally interpolate etc.
- Uniform grids do not catch shadow edges very accurately without consuming lots of memory

Dynamic Lighting

- Dynamic lights are rendered as a separate pass per light
- Only the individual geometry fragments affected by the lights are rendered in the light passes
- All lighting is per pixel, typical light types supported (Point, spot, etc.)
- Lighting model is the same for all assets so they all light the same
- Shadow maps are pre-rendered at the start of the frame, there is an 8mb limit for shadow maps
- Shadow maps are 16-bit linear depth and range controlled for best accuracy

Static occlusion

- Static occlusion is a PVS system
- Database is computed offline using the PS3 to render the scene using queries and the PC to quantize/pack the results – usually an overnight process
- At runtime given a camera position you get a bit array of the visible nodes and the max visible distance
- Each asset fragment belongs to a node and a simple runtime logic op determines if the node is visible
- There is a limited number of nodes, each represented by a single bit

Frame Buffer Setup

■ Display Buffers

- 2 x 1280x720 32-bit RGBA
- Both display buffers share a single tile to reduce wasted memory
- We did not support 1080, couldn't afford the memory

■ Render Buffer

- 1280x720, 32bit, 2xMSAA
- 32-bit depth buffer
- Color and depth in their own tiles with compression enabled
- Always render 1280x720 down sample for NTSC and PAL

Frame Buffer Setup

- Alternate Render Buffer
 - 1280x704, 32bit, 2xMSAA
 - 704 is the magic height as it obeys all the restrictions of depth and color tiles
 - Center on the 720 front buffer, giving 8 pixels of black top and bottom
 - No wasted memory due to alignment. Over 1Mb saved from using 1280x720 and 2% faster
- Idea was too late to be useful on Resistance

Frame Render Order

- Statically lit opaque geometry
- Sky
- Statically lit alpha geometry
- Dynamically lighting passes on opaque geometry
- Dynamically lighting passes on alpha geometry
- Effects, Water, Ground fog
- Resolve multisampling to display buffer, apply color correction (contrast, brightness and saturation)
- Apply post effects on the off screen display buffer
- Draw HUD directly to display
- Draw the system OSD
- Flip the display buffers

Video Memory Budget

- Vram is almost exclusively pixel data, some verts may be in vram for memory balancing
 - 21Mb Frame buffers
 - 512K Fragment programs
 - 8Mb Scratch memory (non-tiled)
 - 50-70Mb Lightmaps
 - 115-135Mb Textures
- Scratch memory is used for shadow buffers, effects, and post effect working buffers etc.

Main Memory

- No CRT allocations
- Allocate our own memory segment
- Allocate all free memory as 1mb pages
- Individually commit pages to the segment
- Map everything in the segment to the RSX
- We do leave a couple of Mb slop for the CRT because its used by other CRT functions (printf) and networking code

Memory Allocators

- Low level allocator is an 'allocate at end' type allocator with no ability to free memory
- The allocator has checkpoints which we can use to rollback main and video memory
- Reliable clean up without concern about destructors being called
- There are checkpoints inserted at the various points during load so we can quickly reload when a player dies or restarts the level
- Systems that needed true dynamic memory dealt with it with an optimal allocator specific to that system

Main Memory Budget

■ Elf+System Setup

- Code and Data 17Mb
 - 2mb of embedded SPU elf data
 - 2+mb of PRX modules
- BSS 8Mb
- CRT Heap 1Mb
- SPU Thread group swap space 2Mb

Main Memory Budget

- Core engine
 - Push buffers 6-10Mb
 - Scratch memory 9Mb
 - Global Textures and HUD 8Mb
 - Misc 2Mb
 - SPU DMA buffers, vertex programs, RT light setup buffers, FIOS init, Sound init etc

Main Memory Budget

■ Level Data

■ Effects	11Mb
■ Geometry	50Mb
■ Collision	9Mb
■ Anim	30Mb
■ Enviroment Instance	2Mb
■ Occlusion/PVS	4Mb
■ Sound	30Mb
■ Dialog	12Mb
■ Physics	9Mb
■ Nav+AI	4Mb
■ Moby Instance	6Mb
■ Gameplay/Scripts	10Mb

SPU Configuration

- 2 Raw mode SPUs
 - One SPU running broad collision
 - One SPU running narrow collision
 - These run all the time
- 3 [Job Manager] SPUs
 - In a thread group running SPURS
 - [Job Manager] policy module
 - All jobs go on these
- 1 Unused
 - This is for the OS to steal for AC3 Encode etc
 - This should be used with its own job manager instance in the future for jobs that don't mind getting interrupted by the OS

SPU Configuration

- Initially we had [Job Manager] using 4 SPUs
- When we finally got a OS that was stealing an SPU it was more expensive to have [Job Manager] on 4 SPUs than on 3
- With AC3 Encode active
 - Total SPU Time for [Job Manager] of 4 4.048 ms
 - Total SPU Time for [Job Manager] of 3 3.924 ms
- Multiple instances of SPURS/[Job Manager] came too late for us to use

SPU Systems

- Animation
- Audio (NextSynth and LR1)
- Bucketer sort
- Collision (separate broad and narrow)
- Dynamic DB
- Dynamic joint
- FX update
- Geom Cull Clip (for shadows and decals)
- Glass
- Moby constants
- Physics collision
- Physics simulation
- Particle (weather fx)
- Render mats
- Static DB
- Water (FFT)

SPUs

- All our systems started off as RAW mode
- The only long term (not finished this frame) asynchronous processing is the collision on the raw SPUs
- We use [Job Manager] but not all systems use it in the typical way of fire and forget. Most of our system require the SPU to be running a particular system at the same time as the PPU.
- To ensure the SPU is doing what we want at the correct time we send [Job Manager] the job and use our own thin synchronization and job buffering schemes using the locked-line for communication
- 10-20% total SPU utilization

Collision Overview

- Dedicated 2 raw SPUs one for broad and narrow phases
- We support immediate and deferred queries
- PPU can directly issue broad or narrow queries
- Broad and narrow overlap, so narrow is processing as soon as the first broad phase result is available
- There are two kinds of deferred collision operations, standard priority which has to done this frame and low priority which has until the end of the next frame
- Immediate requests from the PPU are higher priority than any deferred collision
- Getting game code to optimally use the deferred collision was an issue

Animation Overview

- Approx 400 Joint limit, 15 blended clips
- Full body animations or partials with per joint weights
- Stack driven system similar to ICE
- No blend shapes or set driven key support at present
- While the SPU is computing the animation for a given moby the PPU is updating the next moby and computing its anim stack
- Animation time is typically completely hidden, only time we stall is when waiting for the last anim to complete
- Animation data is compressed in memory and temporarily decompressed before use by SPU

Animation Overview

- Low level animation system is driven by a high level ‘move’ system which builds the anim stack
- End result of the animation is a set of local space matrices
- Subset of skeletal joints called ‘dynamic joints’ which can be modified on post animated data – physics, IK, rag doll, gameplay procedural movement, etc. operate on these
- Any modifications are multiplied back down the hierarchy only for those leaves of the tree by an SPU job
- Finally all resultant local space matrices are sent to another SPU job which builds the world space matrices and pushbuffer which uploads them to RSX
- All skinning is done on the RSX

The End

Questions?