PlayStation® Shader Language for PlayStation®4

Richard Stenson

Chris Ho

US R&D

PS4, PSSL, and Beyond

- Today we will discuss
  - The PS4 architecture
  - Developing for PS4
  - PSSL on PS4
  - Beyond PC with PSSL on PS4
  - Join the discussion

PlayStation®4

- Next Gen PlayStation Console
  - Powerful game machine
  - Modern Graphics features
  - PC based architecture
  - Lightning fast Memory
  - New networking and interface features

Modern GPU

- DirectX 11.2+/OpenGL 4.4 feature set
  - With custom SCE features
- Asynchronous compute architecture
- 800MHz clock, 1.843 TFLOPS
- Greatly expanded shader pipeline compared to PS3™

Fast GDDR5 RAM

- 8GB 256 bit GDDR5
- Fully unified address space
- 176 GB/s total bandwidth
- Massively faster than DDR3
  - 128 bit at ~40GB/s max bandwidth

State of the art CPU

- Modern 64-bit x86 architecture
- 8 cores, 8 HW threads
  - Atomics
  - Threads
  - Fibers
  - ULTs (user-level threads)

GPU+RAM+CPU = Beyond Fast!

- Plenty of power for a true Next Gen Game Experience
  - 8 CPU cores
  - High polygon throughput
  - High pixel performance
  - Efficient branching in GPU Shaders

But what about development?

- PS4 is very approachable for development
  - DX11/OpenGL 4.4 level Shader language in PSSL
  - Powerful Graphics API
  - C++11 CPU Compiler
  - All the expected system libraries and utilities
    - Networking, Codecs, Controllers, Input and more

Familiar PC-like Development Platform

- Full Visual Studio Integration
- Minimal work for good performance
- Built for AAA Games and Indies alike
- Built to enable developers to push the system
  - Good is just the start!
  - Once you are ready for the deep dive we support you there as well

What is PSSL

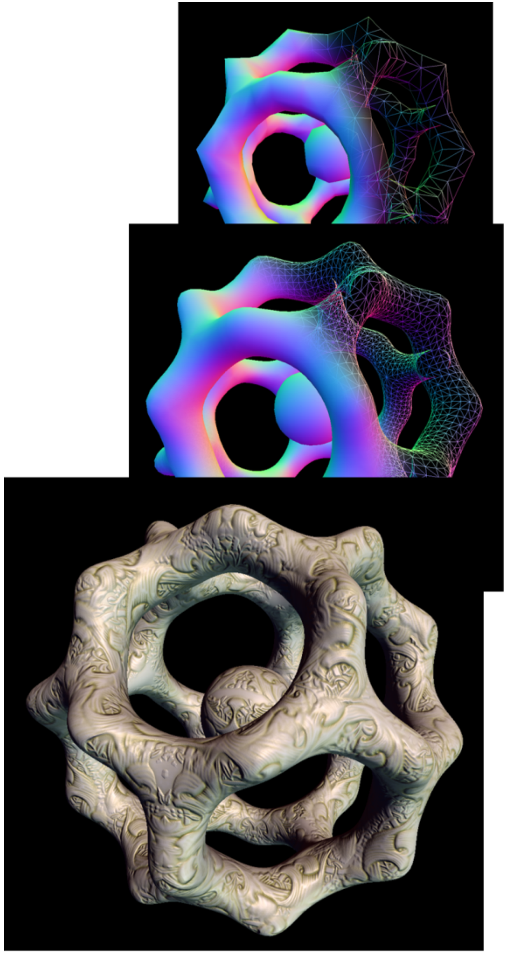
- PSSL is the PlayStation Shader Language for PS4
- Supports modern graphics development
  - Vertex
  - Pixel
  - Geometry
  - Hull
  - Domain
  - Compute

Vertex and Pixel Shaders

- Next generation VS and PS Shaders
- Extended support based on our hardware
  - RW_Textures and Atomics in all shaders

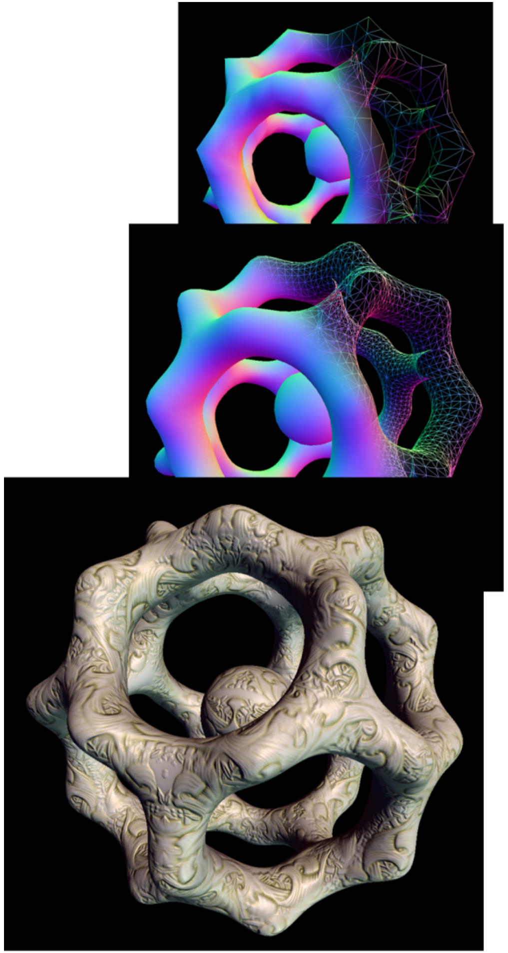
Geometry Shaders

- Supports special cases GS like
  - GS Tessellation
  - Instancing
  - Cube mapping
  - Streamout



Hull, and Domain

- Supports HS DS Tessellation
  - Parametric surface conversion
  - Optimal Geometry generation



Compute

- Support modern compute shaders
  - Parallel Multithreaded execution
  - This cross wave and group synchronization primitives like barriers and atomics
  - Various Local and Global memory pools for complex thread interaction

What does PSSL look like?

- It follows the PC conventions for shaders
- ANSI C style syntax and coding rules
- Includes the expected:
  - Vectors
  - Standard libs
  - C++ style structs with members
  - Supports static and dynamic control flow

A simple vertex shader

```
struct VS_INPUT
{
    float3 Position     : POSITION;
    float3 Normal       : NORMAL;
    float4 Tangent     : TEXCOORD0;
    float2 TextureUV    : TEXCOORD1;
};

VS_OUTPUT main( VS_INPUT input )
{
    VS_OUTPUT Output;

    Output.Position = mul( float4(input.Position.xyz,1), m_modelViewProjection );
    …
    float3 vN = normalize(mul(float4(input.Normal,0), m_modelView).xyz);
    …
    return Output;
}
```

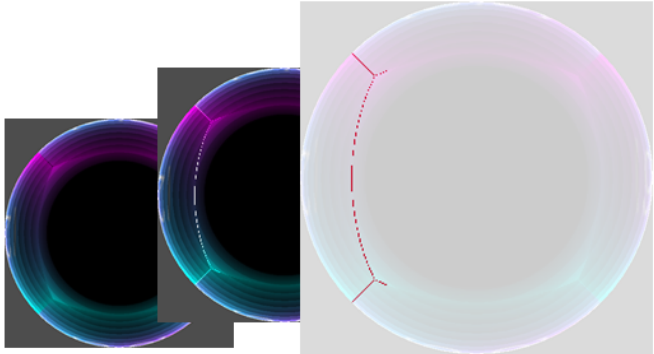
A simple pixel shader

```
SamplerState samp0 : register(s0);
Texture2D colorMap : register( t0 );
Texture2D bumpGlossMap : register( t1 );

float4 main( VS_OUTPUT In ) : S_TARGET_OUTPUT
{
    ...
    float4 diff_col = colorMap.Sample(samp0, In.TextureUV.xy);
    float3 spec_col = 0.4*normalGloss.w+0.1;
    ...
    return float4(vLight.xyz, diff_col.a);
}
```

How PSSL is being developed

- World wide collaborative efforts
  - US R&D Shader Technology Group
  - PS Vita shader compiler team in ATG
  - Graphics driver team in ICE
  - GPU hardware teams and SDK managers
  - With tight feedback with Sony World Wide Studios
- QA Team
  - Thousands of automated tests



Let's see some PSSL shaders in action

- This is real-time PS4 game footage
- All shaders in these demos were built with the PSSL tool chain

Porting to PSSL from the PC

- Easy initial port target
  - Simple conversion of your PC or Xbox 360 Shader
  - PS3 Cg conversion is fairly trivial
- Prototyping on the PC much simpler this generation

Maintaining PSSL and PC Shaders

- Simpler to maintain code this round
  - PC and PS4 are now much closer for shaders
  - All of the shader stages and features are available in PSSL
    - Often have been extended
- This means you should be up and running very quickly
  - The time to “my first tri” will be better
  - The time to “my game runs!” will be better
  - The time to “my game is fast on PS4” will also be better!

Beyond PC with PSSL on PS4

- Extended Buffer Support for all shaders
  - Not just Pixel and Compute
  - The hardware is capable so we expose that.
- Special Hardware Intrinsics
  - Some native ISA instructions are natively supported
    - ballot - Good for fine grain Compute control
    - sad - For multimedia tasks like Motion Estimation for accelerated image processing

Beyond modern PC shader features

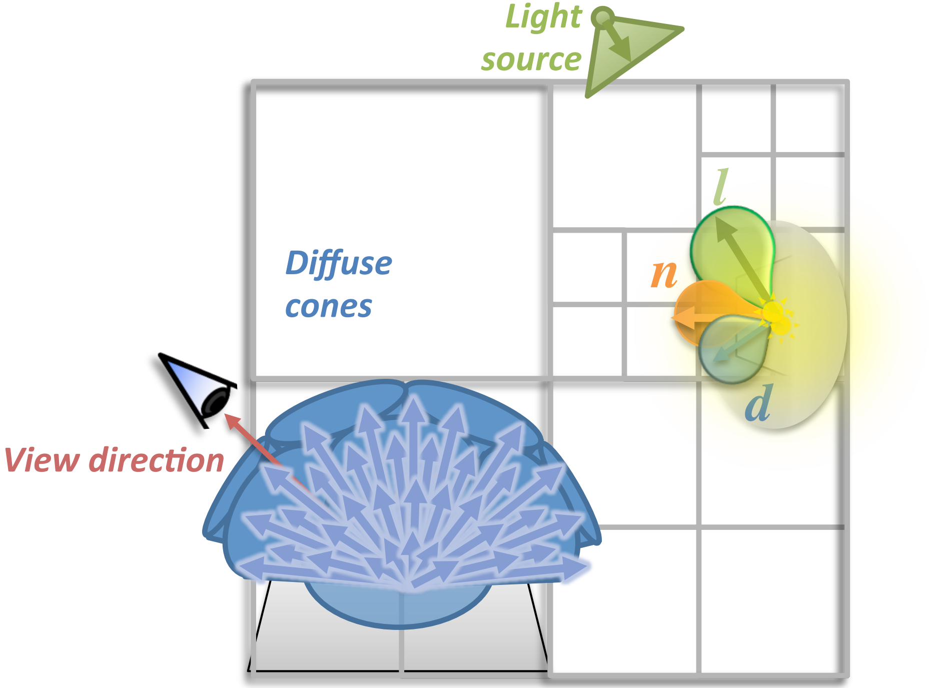
- PS4 GPU has many special features
- Let's talk about a specific example

Example

- New features over previous generation
  - New shader stages
    - Hull, Domain, Geometry, Compute
  - Atomics and RW_Buffers
    - Accessible in all stages
  - Partially Resident Textures
- What can we do with all of this?
  - Why not Sparse Voxel Octree Cone Tracing!

Sparse Voxel Octree Cone Tracing

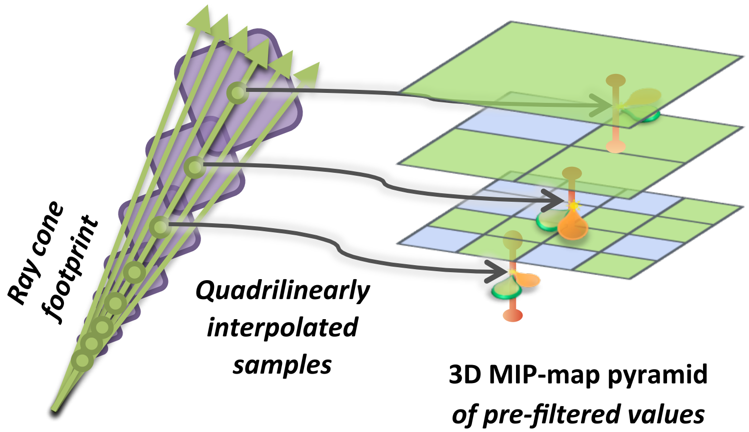
- Global Illumination solution proposed by Crassin et al. in 2011
- Trace cones through a voxelization of the scene to solve for the contribution of direct and indirect light sources



Images credit Cyril Crassin's GTC presentation "Octree-Based Sparse Voxelization for Real-Time Global Illumination"

Sparse Voxel Octree Cone Tracing

- Prepass: voxelize static geometry
- During gameplay:
  1. Voxelize dynamic geometry
  2. Light volume
  3. Build mipmaps
  4. Render gbuffers
  5. Cone trace scene



Images credit Cyril Crassin's GTC presentation "Octree-Based Sparse Voxelization for Real-Time Global Illumination"

Sparse Voxel Octree Cone Tracing

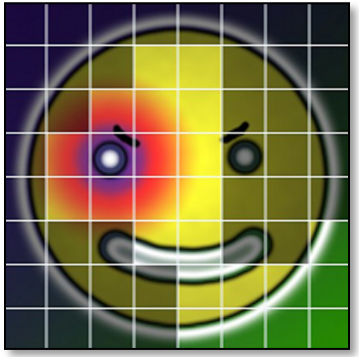
- Could do a full implementation
  - (RW_)Texture3D for bricks
  - (RW_)RegularBuffer for octree representation
  - Geometry shader for thin surface voxelization
- Other useful PSSL features
  - Partially Resident Textures?

Partially Resident Textures

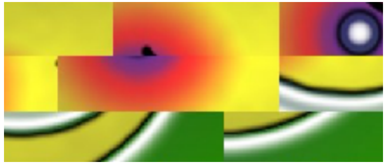
- Also called “Tiled Resources”
- Hardware Virtual Texturing
- Textures broken up into 64KiB tiles
- Tile texel dimensions dependent on texture dimensionality and underlying texture format
- Allows for not all the texture to resident in memory at a time

Partially Resident Textures

- Like this, but in hardware!



Virtual Texture



Physical Representation

- For more information, please refer to the Hardware Virtual Texturing slides presented at SIGGRAPH 2013

PSSL and PRT

- Exposed in PSSL as a new Sparse_Texture* type
  - All sample-able texture types supported, 1D, 2D, 3D, Cube, Arrays, etc.
- Sample() modified to take an extra `out` parameter to indicate status
- It's not necessary to use the Sparse_Texture type to utilize partially resident textures, but Sparse_Texture is necessary if you want status information!
  - Essentially page-fault tolerant GPU memory accesses

PSSL Sample Code

```
Sparse_Texture2D<float4> sparseTexture;

float4 main(VS_OUT inv) : S_TARGET_OUTPUT0
{
    SparseTextureStatus status;
    float4 sampleColor;

    // Try the regular LOD level first
    sampleColor = sparseTexture.Sample(status.code, sampler1, inv.tex0);

    // If 'Sample' fails, handle failure
    if ( status.isTexelAbsent() )
        …
```

SparseTextureStatus

```
struct SparseTextureStatus
{
    uint code;

    bool isTexelAbsent();
    bool isLodWarning();
    uint getAbsentLod(); // LOD of absent texel
    uint getWarningLod(); // LOD that caused the warning
};
```

PRT Applications

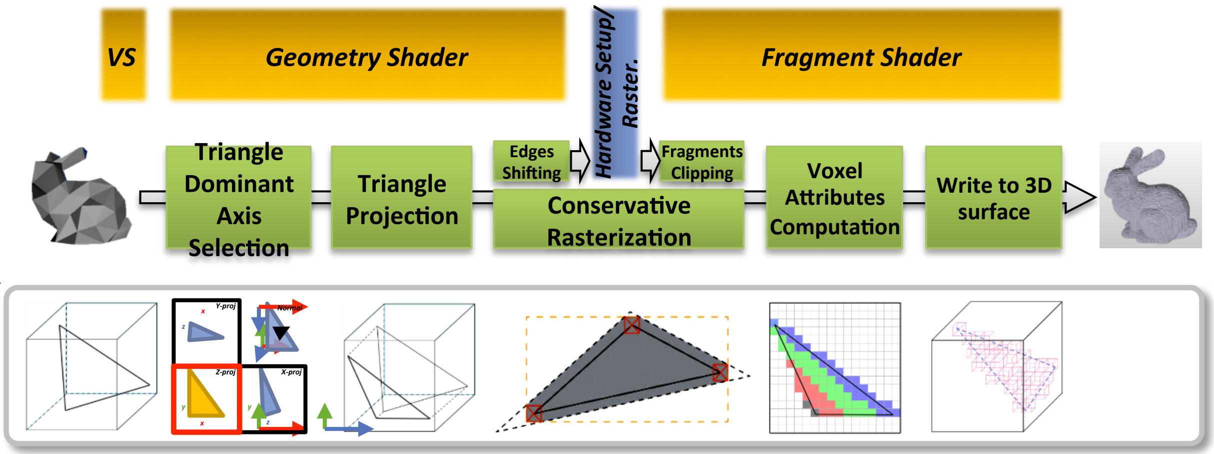
- Megatexturing
- Ptex
- Sparse Voxel Cone Tracing

Sparse Voxel ~~Octree~~ Cone Tracing

- Instead of populating an octree, use a partially resident texture!
- Pros:
  - PRT tiles do not need to be padded for proper interpolation
  - No need to build an octree data structure
  - No need to incur the indirection costs of traversing an octree data structure
- Cons:
  - PRT tile dimensions not ideal – 64x64x4 for 32-bit 3D textures
  - No fast empty space skip from octree traversal

Voxelization

- Adaptation of Crassin's method detailed in OpenGL Insights
  - Unfortunately no atomic floats; quantized ints for accumulation rather than spin lock
- Use geometry shader and hardware rasterizer to voxelize scene into a 3D texture with a single pass



Images credit Cyril Crassin's GTC presentation "Octree-Based Sparse Voxelization for Real-Time Global Illumination"

PUBLIC
Sony Computer Entertainment

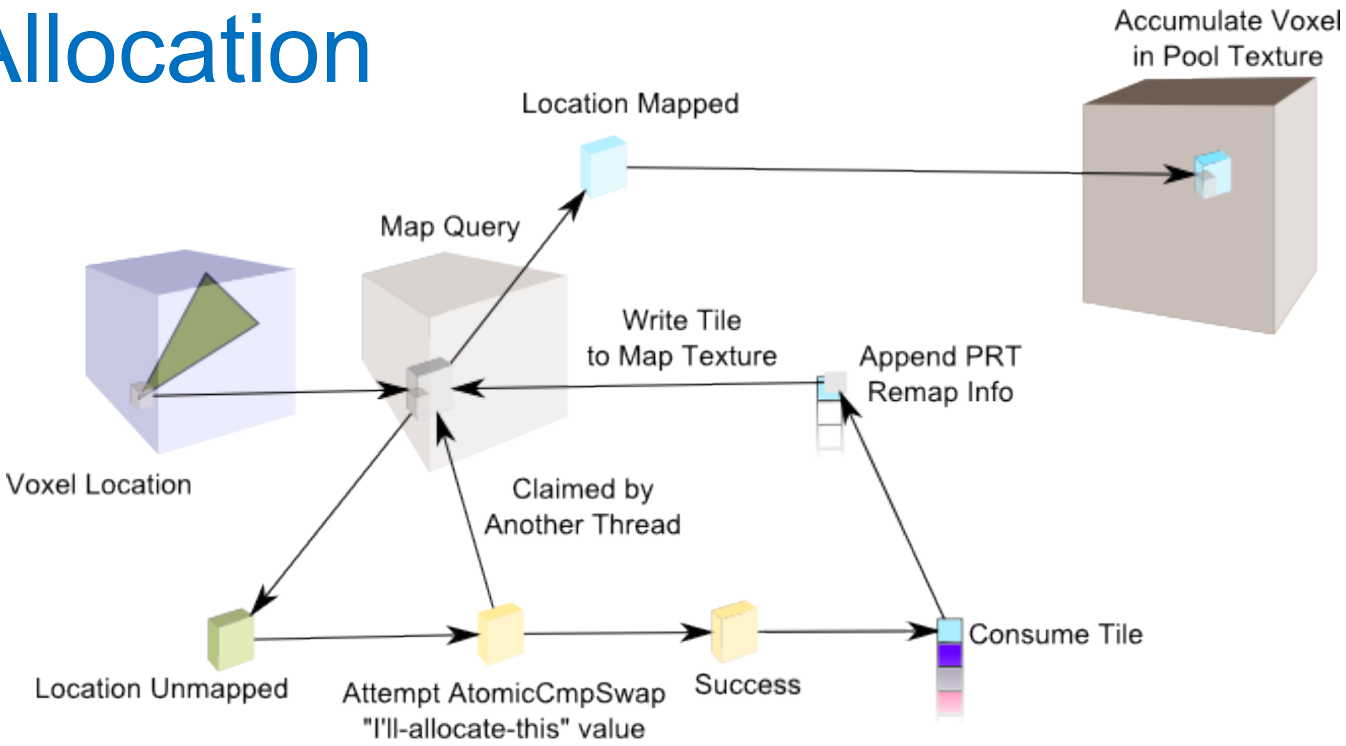
Writing to a empty Sparse Texture?

- Problem: the texture is unmapped to begin with!
  - No pages are mapped yet, can't write to memory that doesn't exist!
- Idea: write to the pool texture instead
  - PRT allow us to map the same physical page to multiple virtual locations
  - All tiles are mapped into the pool texture and then doubly mapped to the sparse texture as need
- Fragments that need to be written out query a map texture before writing, and if the tile is ummapped they allocate a tile and write it back to the map texture
  - Keep free tiles in a Consume buffer, write out re-map info into an Append buffer

Tile Allocation

- Map texture initialized to set a reserved “unallocated bit”
- AtomicCmpExchange() in a value to flip on an additional “unallocated-but-I’m-working-on-it” bit for a single thread
  - Consume() a free tile
  - Append() consumed tile with remap data
  - Write out tile location to map texture
- Write into the tile using pool texture
- After pass completion, read from append buffer on CPU side to map tiles from the pool to the sparse texture

Tile Allocation



Tile Allocation

```
const uint unallocated = 0x80000000, allocating = 0xC0000000;
do {
  cur = map[tileLoc];
  if(cur == unallocated) {
    uint output = 0xffffffff;
    AtomicCmpExchange(map[tileLoc], unallocated, allocating, output);
    if(output == unallocated) {
      cur = g_freeTiles.Consume();
      map[tileLoc] = cur;
      g_remaps.Append(…);
    }
  }
}
while(cur & unallocated);
```

Implementation

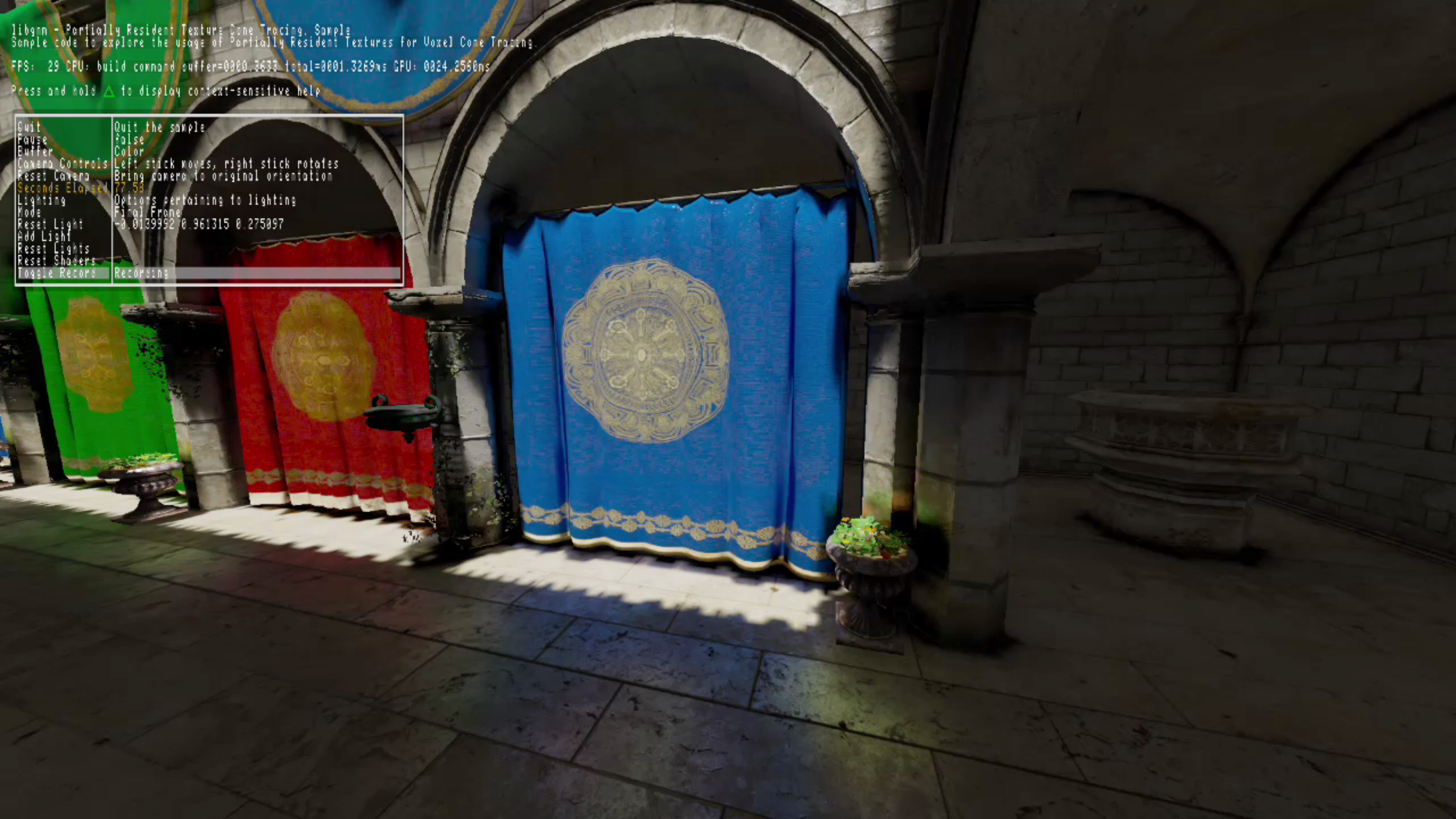
- 1024x1024x512 32-bit pool texture
  - 16x16x128 tiles, given linear ids (can use shifts/masks to find actual location)
- 512x512x512 32-bit Sparse Texture to represent the scene
- 8x8x128 map texture for tile allocation
- Consume buffer for grabbing free tiles
- Append buffer for noting allocated tiles for remapping

Building Mipmaps

- Compute Kernel that takes an 8x8x8 brick and reduces it to a 4x4x4 brick
  - LDS for accumulating final values
- Allocate tiles for new mips in the same manner as voxelization
- Pre-map the lowest mips (all that fit into 64KiB)

Lighting Voxels

- Currently naively lit
- Spawn Compute kernel for entire 3D texture, iterate over lights if resident
- Needs optimization


libgnm - Partially Resident Texture Cone Tracing. Sample
Sample code to explore the usage of Partially Resident Textures for Voxel Cone Tracing.
FPS: 29 CPU: build command buffer=0000.3633 total=0001.3269ms GPU: 0024.2560ms
Press and hold △ to display context-sensitive help
Quit Quit the sample
Pause false
Buffer Color
Camera Controls Left stick moves, right stick rotates
Reset Camera Bring camera to original orientation
Seconds Elapsed 77.53
Lighting Options pertaining to lighting
Mode Final Frame
Reset Light -0.0139992 0.961315 0.275097
Add Light
Reset Lights
Reset Shaders
Toggle Record Recording

Results

- Average frame time: 26ms
  - 3ms gbuffer, 11ms indirect + specular reflect, 11ms direct
- Memory usage:
  - 2GiB Pool Texture, ~315MiB allocated after voxelization, ~56% resident
- Static geometry voxelization and lighting time:
  - 45ms voxelization, 22ms top-mip light, 25ms mip regeneration
- Still much more optimization possible!

PSSL is still evolving

- Features in consideration:
  - Control of shader resource layout
  - More exotic compute primitives for GPGPU
  - Tightly coupled Graphics and Compute
  - And many more…

Join the discussion

- We would like to hear from you!
- Sign up as a PS4 developer, if you're not already
  - http://us.playstation.com/develop/
  - There is a link for all territories from this page
- We are looking for solid suggestions with clear benefits
  - Specific performance benefit
  - Special new/novel feature, etc.

Push the boundaries with PSSL

- PS4 is a powerful, but friendly to develop for
- PSSL is one of the keys for developing for PS4
- Our goals with PSSL
  - Make better Games
  - Push the boundaries on PS4
  - And to be efficient in that process

- Help us help YOU!

Q&A

- Questions?

US R&D Shader Technology
Group is hiring!