Blender Cycles

Architecture of a unified CPU/GPU path tracer

Sergey Sharybin

Graphics Programming Conference, November 18-20, Breda 2025

But first...

Blender 5.0 is out!

Massive release, check out at blender.org

Talk Overview

- Introduction
  - What is Cycles?
  - Difference from real-time engines
- Architecture
  - Overview
  - Render scheduling
- Kernel
  - Unified kernel for all GPGPU backends
  - Megakernel vs. wavefront render
  - Kernel scheduling
- Deeper dive
  - Shading Virtual Machine
  - Texture filtering
  - Interactive viewport
- Q&A

Introduction

What is Cycles?

- Physically-based path tracer for production rendering
- Developed by the Blender project under the Apache 2 license
- Render engine that is interactive and easy to use
- Supports many production features
- Uses physically based shading system
- Design goals include handling large amount of geometry and complex shading and lighting scenarios

Cycles features

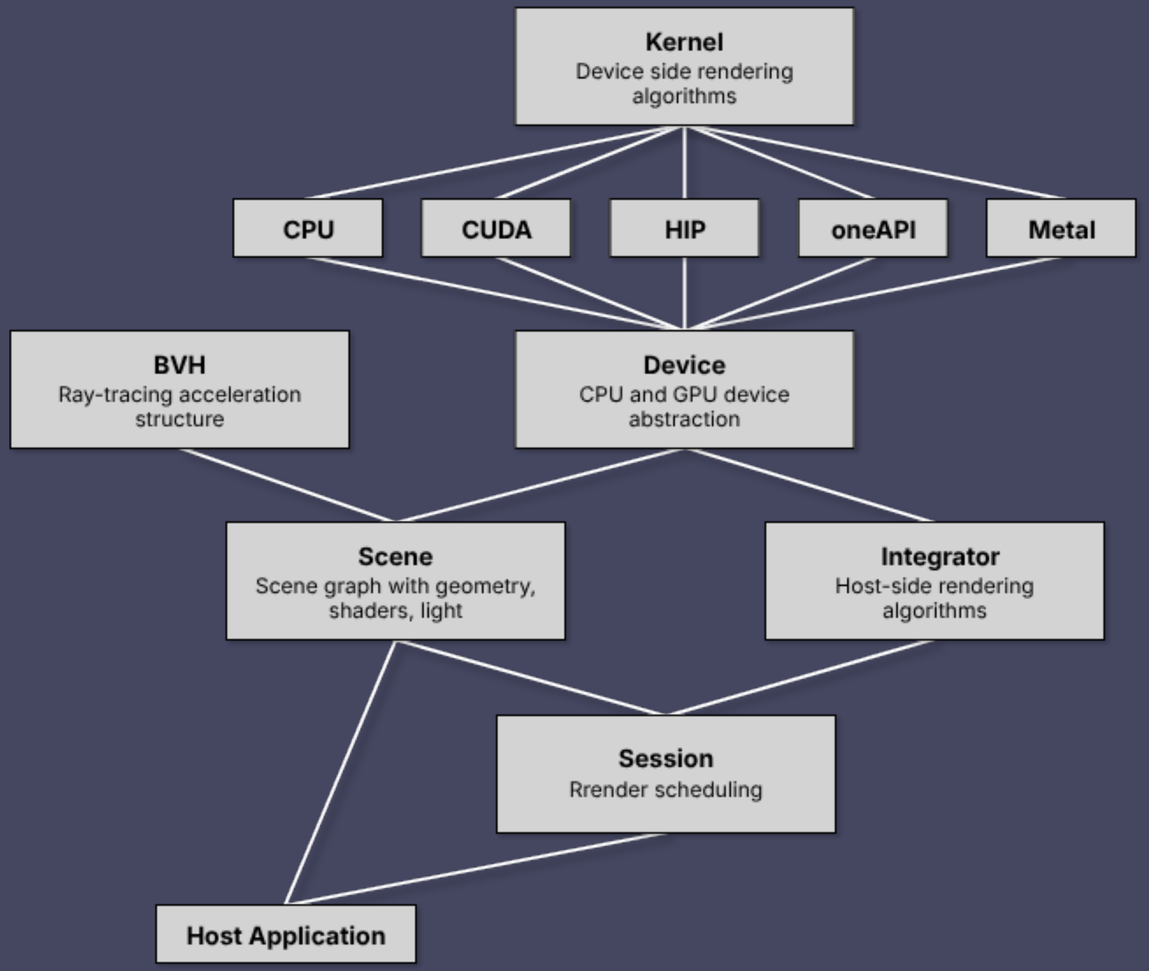
- Unified rendering kernel for CPU and GPU (AMD, Apple, Intel, Nvidia)
- Cross platform and multi-device support (even heterogenous)
- Compositing (render layers and passes, shadow catcher, etc.)
- Denoising, adaptive subdivision, camera lens models
- Shading: node-based, OpenBPR compatible (mostly), OpenShadingLanguage (OSL)
- Light: global illumination, light portals, light and shadow linking, manifold next event estimation (MNEE), path guiding
- Many more!

Difference from real-time engines

- Path tracing render engine
  - Takes long time to converge, but allows for complex light interactions
  - Implements full global illumination
- Shading Virtual Machine (SVM)
  - Byte-code machine which interprets shader node graph
- Kernel is written in C++
- Matched features set on CPU, GPU, and GPU+HWRT

Architecture overview

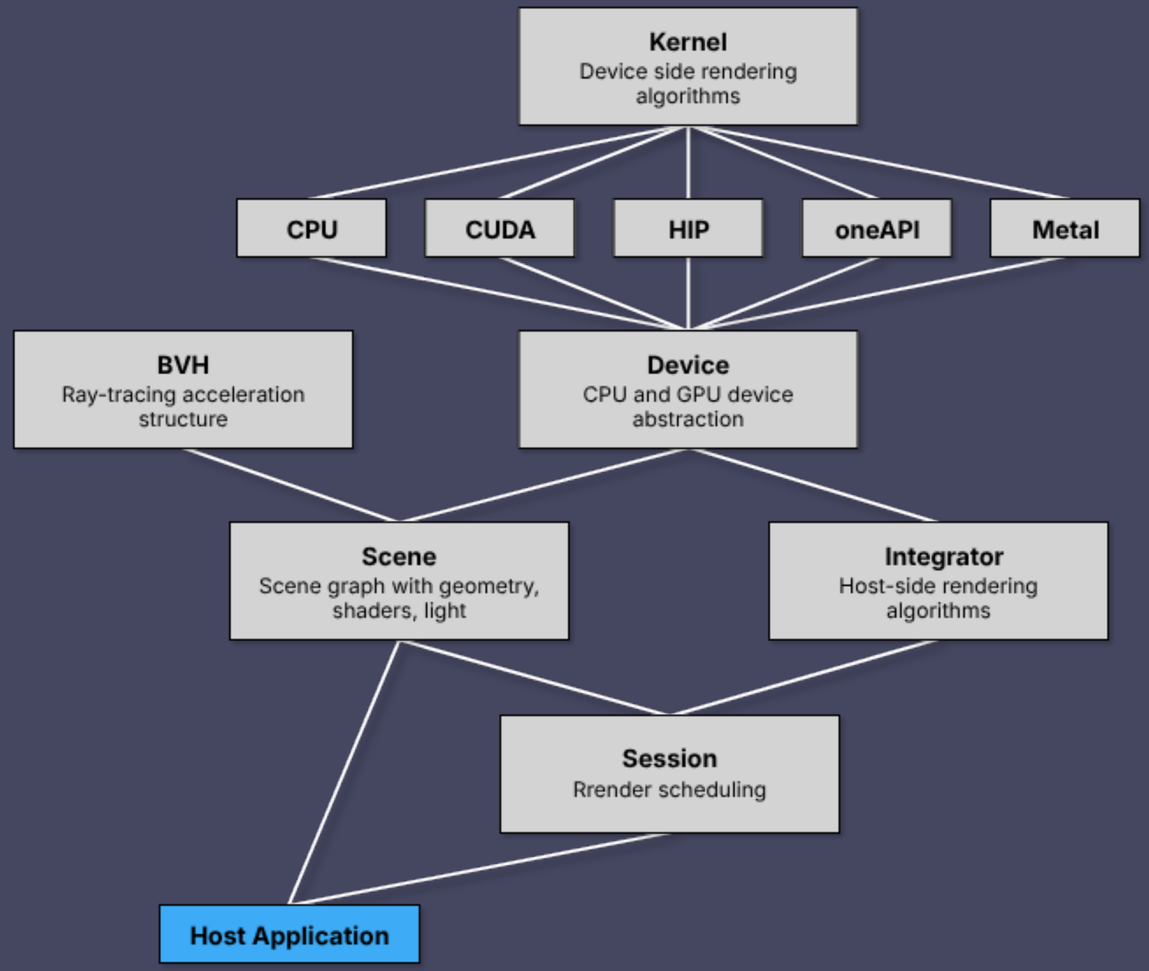
Architecture overview



Architecture overview

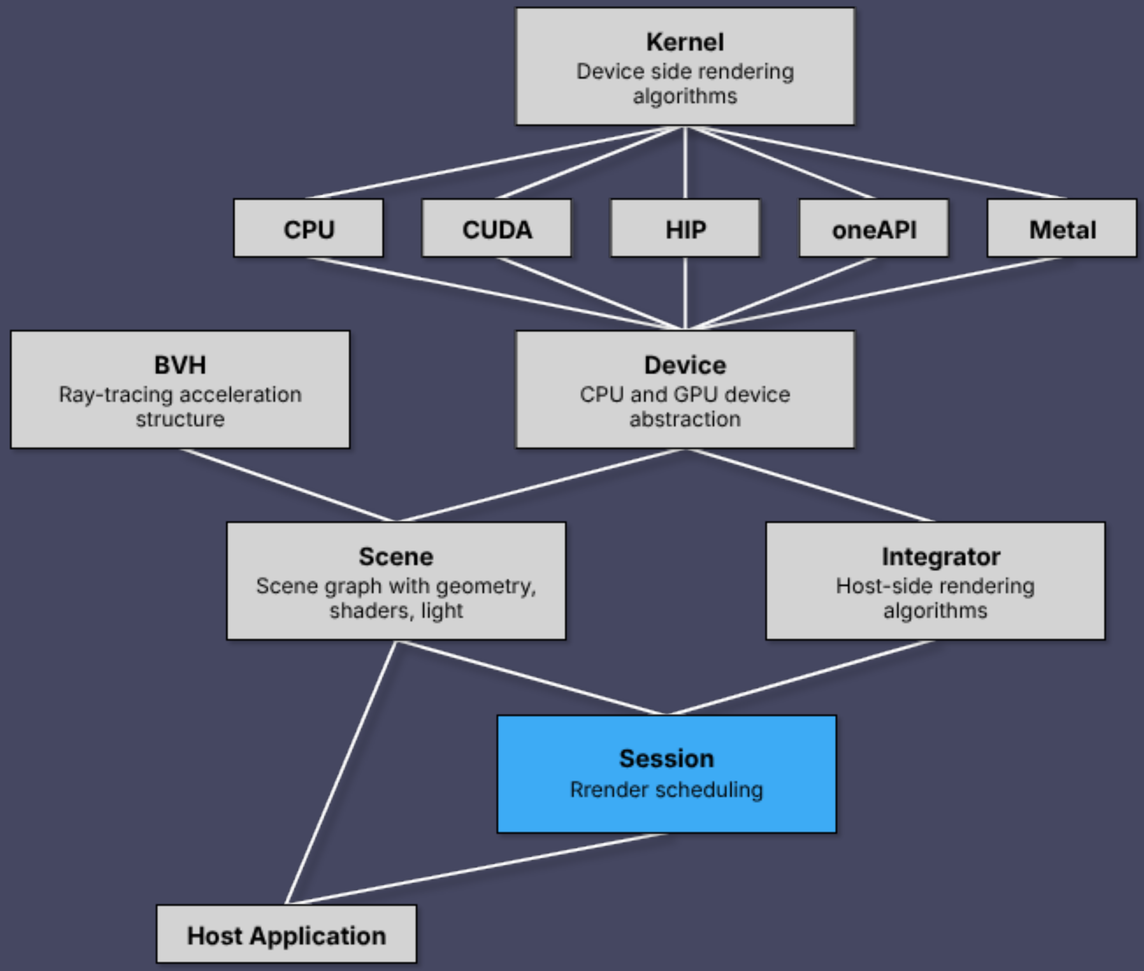
Host application adapter

- Takes care of converting data from DCC to Cycles scene graph
- Its responsibility only synchronize changes from the host application
- Adapter from Blender is maintained by the Foundation
- There are external integrations (Rhino3D, Cinema 4D, etc.)
- Debugging tool: XML reader for Cycles standalone



Architecture overview
Session

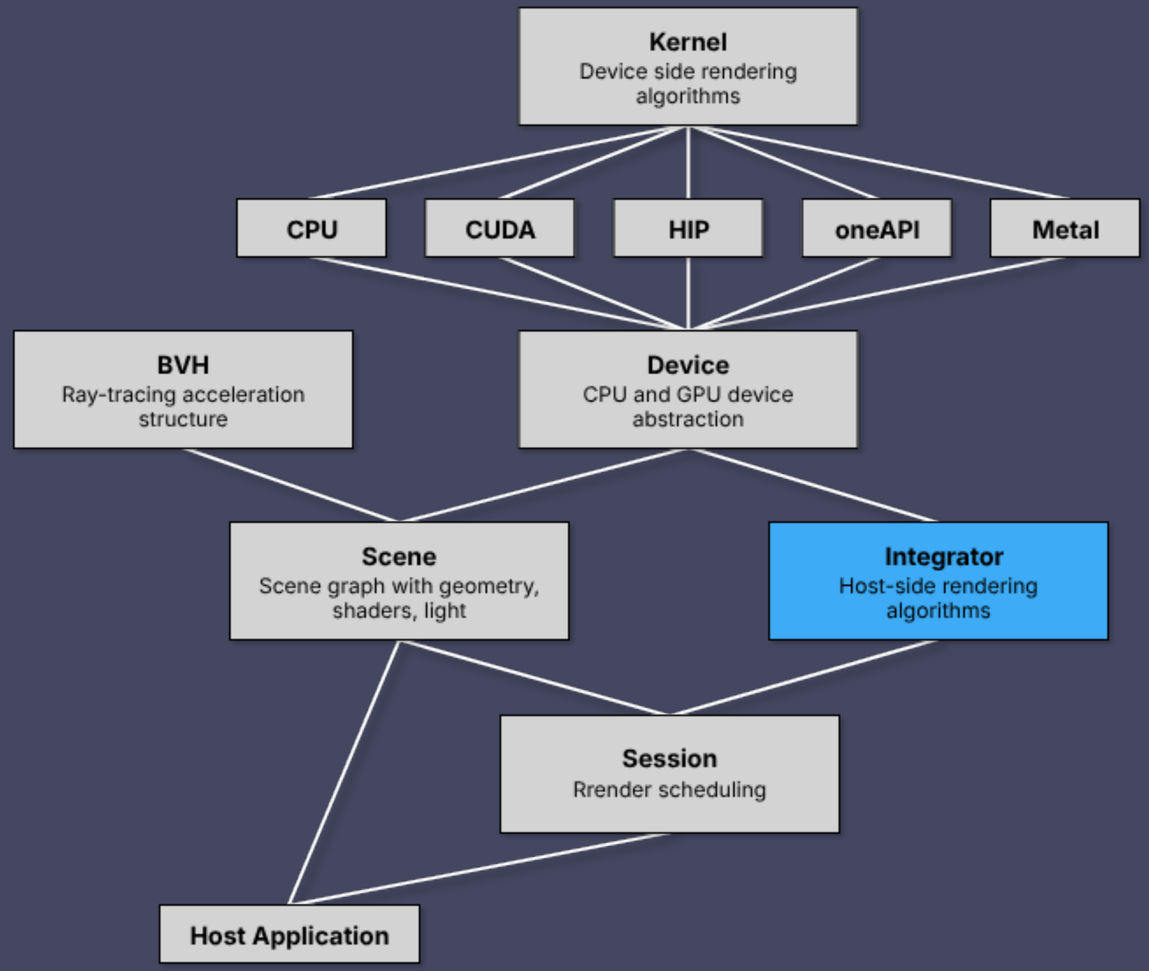
- Root object created by the host application
- Owns all data needed for rendering
  - Device abstraction object
  - Display and write drivers
  - Scene graph
  - Render buffer
- Render scheduling



Architecture overview

Integrator

- Host-side algorithms for path tracing integrator
- Dynamically schedules and invokes kernels, maintaining wavefront
- Balances work between multiple devices
- Integrates with denoising and adaptive sampling algorithms
- Provides accessors to passes in render buffers

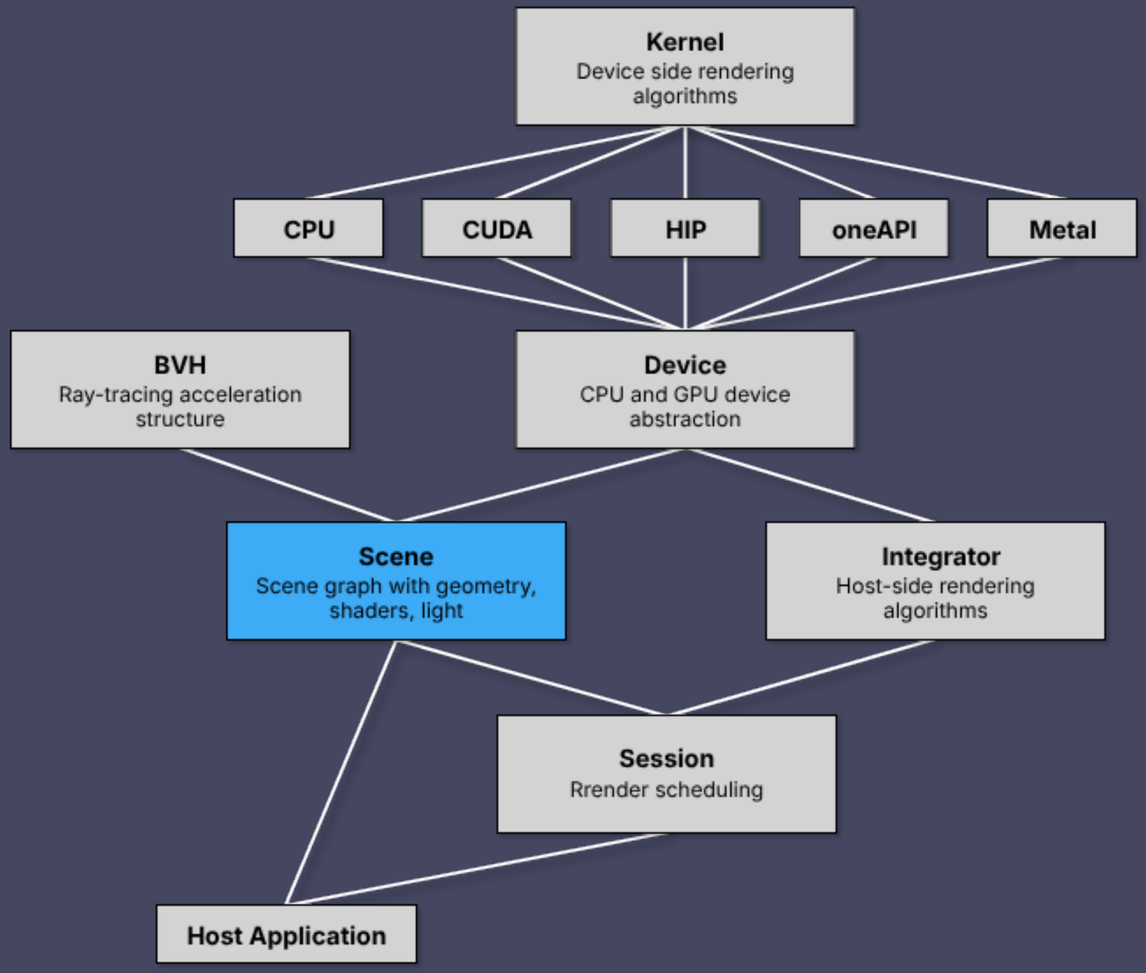


Architecture overview
Scene

Render scene data, stored as nodes of a scene graph

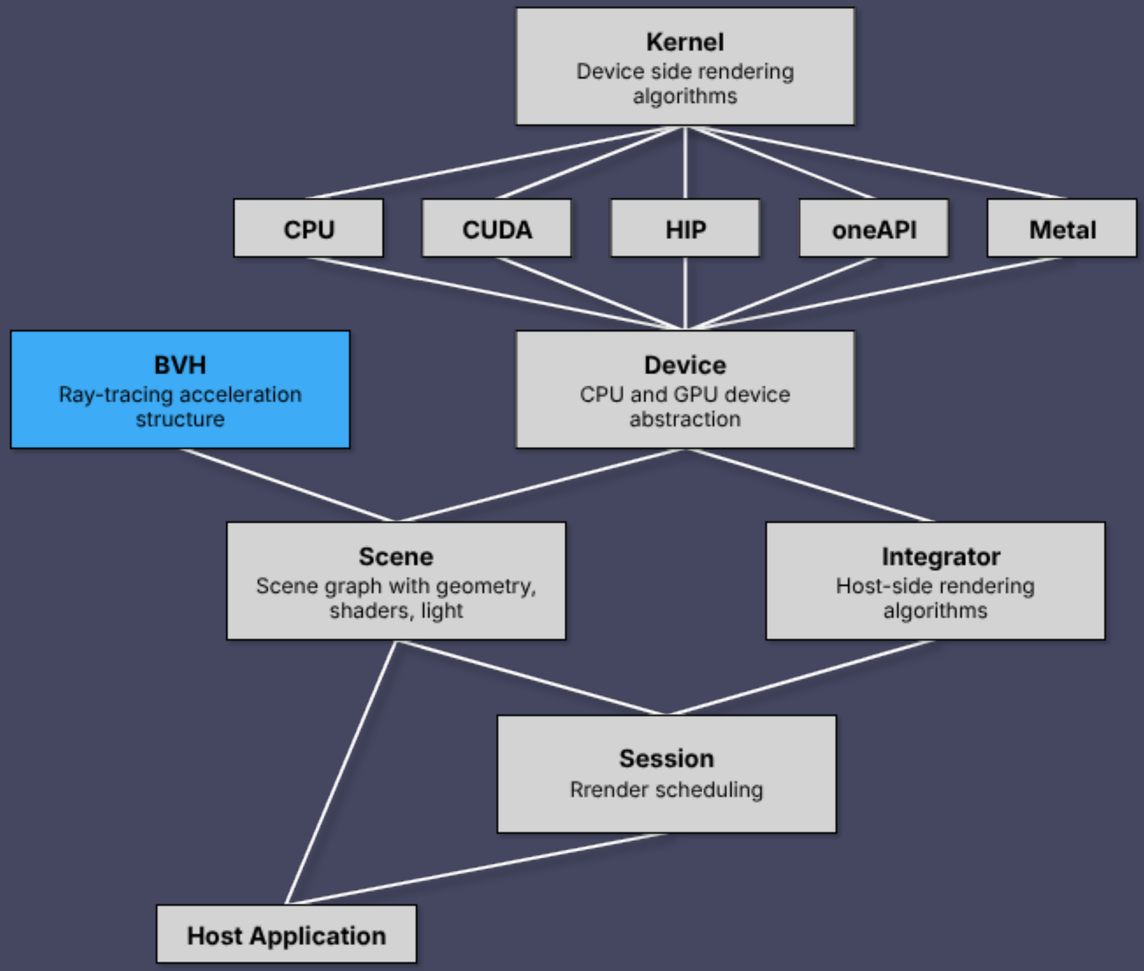
Everything is a node:

- Scene itself is a node
- Geometry
- Integrator, film settings, etc.
- Shading graphs



Architecture overview
Bounding Volume Hierarchy (BVH)

- BVH is a ray-tracing optimization tree structure
- Cycles supports multiple acceleration structures
  - Embree for CPUs and Intel GPUs
  - OptiX BVH for hardware ray-tracing on NVIDIA GPUs
  - HIP RT for hardware ray-tracing on AMD GPUs
  - Metal RT BVH for hardware ray tracing on macOS
  - Custom BVH2 implementation for everything else
- Viewport updates use BVH refitting to keep up interactivity

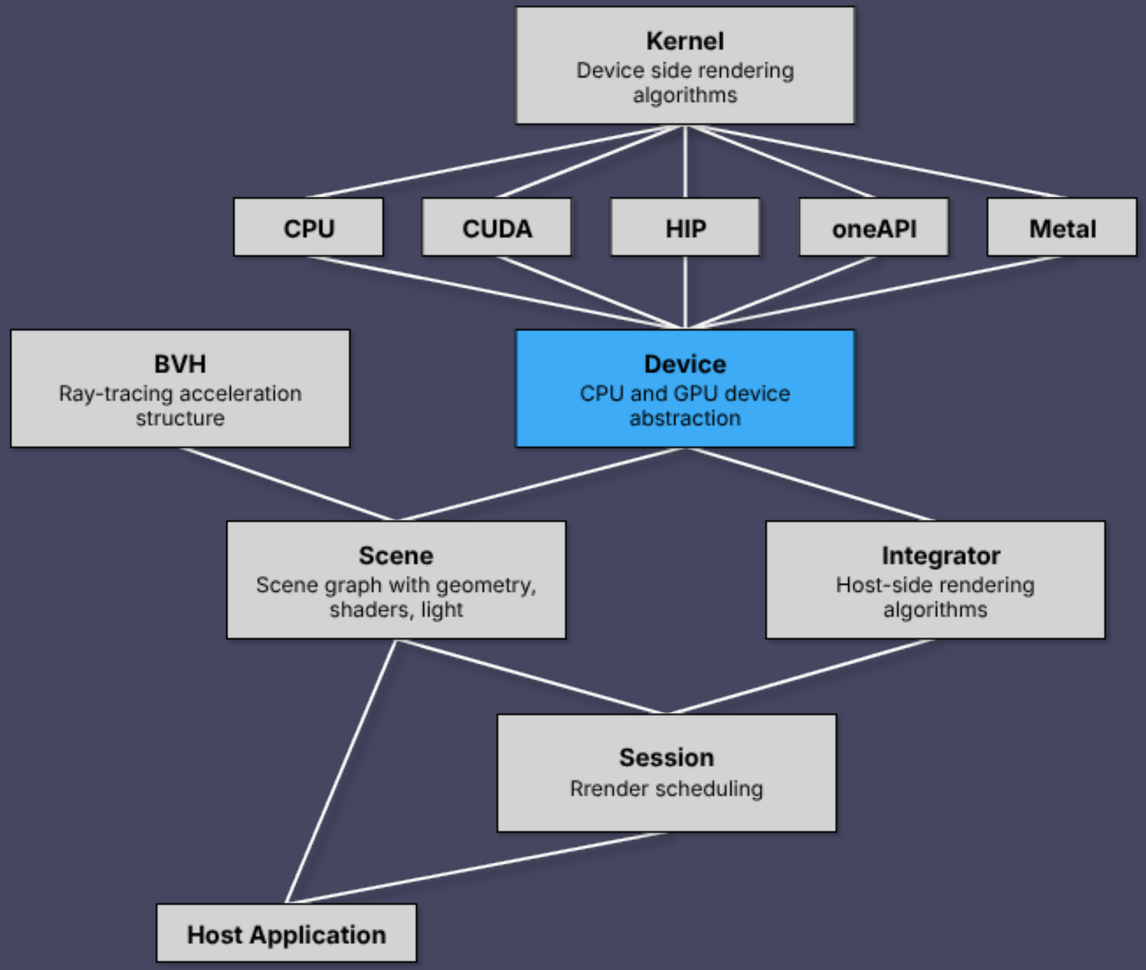


Architecture overview
Device

An abstraction of compute backend to allow render algorithms manipulate memory and call functions (invoke kernels).

Supports a variety of backends

- CPU
- GPU: CUDA, OptiX, HIP, HIP RT, oneAPI, Metal
- Multi: GPU+GPU or CPU+GPU



Render scheduling
Main steps

- Allocate and clear render buffers
- Render loop:
  - Render a number of samples on devices
  - Rebalance work between devices
  - Adaptive sampling error redistribution and stopping
  - Render buffer postprocessing
  - Denoising
  - Display texture update

Kernel

Kernel
Language overview

- Single kernel source compiled for all backends
  - There is backend specific logic for hardware raytracing and texture sampling
- Different entry points for CPU and GPU backends
  - Entry points are tiny wrappers
  - All GPU backends use the same code for entry points
    Exception: hardware raytracing

Kernel

Language overview

The code uses a subset of C++17, CUDA, and HIP:

- Need to be careful to only use language features that are supported for all targets
- Preprocessor macros are used to smooth over the language differences
  For example address space qualifiers ccl_global, ccl_local, etc.
- No recursive functions, function pointer, dynamic memory allocation, no doubles
- Some utility functions needed to be re-implemented

Kernel

Vector types

- The vector types are the same as CUDA: float2, float3, float4
  Similar for int, uint, uchar
- Operators like add, multiply, etc. work as expected
- Construction is a bit different: make_* function is used:

```
float3 v = make_float3(x: 0.0f, y: 1.0f, z: 0.0f);
```

- Necessary classes and operator overloads are implemented to support all platforms

Kernel

Constant memory and textures

- Small, fixed size data is stored in constant memory.
  KernelData kernel_data contains all constant memory, and is available as a global variable everywhere

- All large read-only data is stored in a small number of arrays in global memory, and texture handles for image textures

For historical reasons this is still called "textures", as old GPU architectures had to use texture memory for good performance

Kernel

CPU SIMD optimization

- Math utilities heavily utilizes SIMD intrinsics
  - There is a fallback "naive" scalar implementation
- Mainly utilizes SSE2, SSE4.1, AVX, and AVX2 intrinsics
- ARM64 support heavily relies on sse2neon.h
  - sse2neon is pretty good, but sometimes has undesirable overhead
  - Such cases are identified on case-by-case basis, and native ARM64 Neon intrinsics are used

Kernel
Megakernel

- Megakernels compute a light path from start to end
- Simple to implement
- Not efficient on GPU for a production renderer
- Turns out to be the most efficient on the CPU
  - At least without using wide SIMD for shading
  - Or until ray packs are used

```
// Path tracing megakernel pseudo-code.
Spectrum trace_path(int image_x, int image_y, int max_bounce)
{
  Ray ray = generate_ray(image_x, image_y);
  Spectrum result = zero_spectrum();
  for (int bounce = 0; bounce < max_bounce; bounce++) {
    const Hit hit = scene_intersect(ray);
    if (!hit) {
      break;
    }
    result += evaluate_emission(hit);
    result += evaluate_surface(hit);
    ray = surface_bounce(ray, hit);
  }
  return result;
}
```

Kernel
Megakernel

Cycles implements megakernel approach on CPU

```
ccl_device void integrator_megakernel() {
  while (has_scheduled_kernels()) {
    // Handle any shadow paths before we potentially create more shadow paths.
    integrator_intersect_shadow();
    integrator_shade_shadow();

    // Handle regular path kernels.
    integrator_intersect_closest();
    integrator_shade_background();
    // ...
  }
}
```

Disclaimer: the code is simplified for the demonstration purposes :)

Kernel
CPU scheduling

- CPU rendering traces a light path from start to end in each thread
- Multi-threading uses a simple parallel for loop over all pixels and samples to be rendered
- A single megakernel calls the individual microkernels as needed, sharing code with the GPU implementation

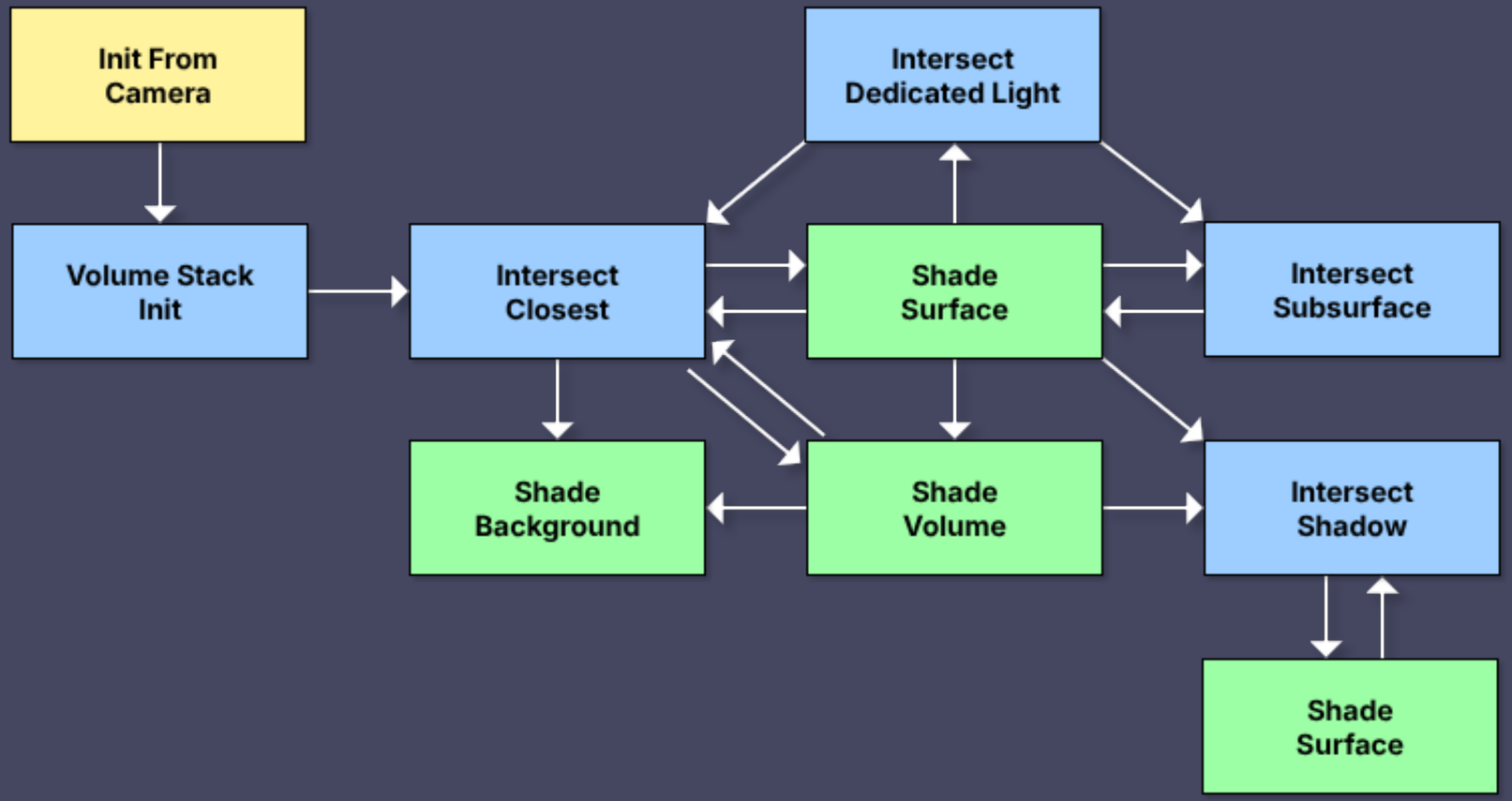
```
tbb::task_arena local_arena = tbb::task_arena(num_cpu_threads);
local_arena.execute(f: [&]() ->void  {
  tbb::parallel_for(first: int64_t(0),  last: num_pixels,
    f: [&](const int64_t work_index) ->void  {
      KernelWorkTile work_tile;
      work_tile.y = work_index / image_width;
      work_tile.x = work_index - work_tile.y * image_width;
      // ...
      for (int sample = 0; sample < num_samples; sample++) {
        integrator_megakernel(kernel_globals, state, render_buffer);
      }
    });
});
```

Kernel
Wavefront

- Cycles uses wavefront path tracing on the GPU:
  - Megakernels Considered Harmful: Wavefront Path Tracing on GPUs
  - The Iray Light Transport Simulation and Rendering System
- The problem it solves: threads coherency
- The idea is that there are individual kernels for each task: scene intersect, surface shading, etc.

Kernel

Scheduling graph



Kernel

Wavefront integrator state

- The state of each path is stored in an IntegratorState
- This state contains all information for the following kernels to compute the rest of the path
- Memory is reserved for millions such integrator states
- Each integrator state can be active or inactive. If it is active, it stores the next kernel to be executed
- Macros are used to abstract state access:

```
const float min_ray_pdf = INTEGRATOR_STATE(state, path, min_ray_pdf);
INTEGRATOR_STATE_WRITE(state, path, min_ray_pdf) = fminf(unguided_bsdf_pdf, min_ray_pdf);
```

Kernel

Structure of arrays

- On the GPU, structure-of-arrays storage is used
- Generally for more efficient memory access patterns
- There are exceptions when using array-of-structures is preferred
  - For example, ray intersection data which needs to be fully read by every kernel
  - Structure-of-Arrays-of-Structures :)
- The state is code-generated using macros

Kernel
Scheduling

```
Mark all paths as inactive


While any work tiles remain to be rendered:
    If fewer than half of paths are active:
        Get next tiles to be rendered
        Gather array of inactive path indices
        Execute init_from_camera kernel to activate inactive paths


    Find the kernel that most active paths need to execute next
    Gather array of active path indices with this kernel
    Execute kernel
```

Kernel
Logistics

- We prefer to bundle pre-compiled kernel binaries
  - We ship CUDA, HIP, HIP-RT, oneAPI kernel binaries
  - OptiX kernel is shipped as PTX: requires one-time client-side optimization
  - Metal is always JIT :(
- To minimize the distribution size the binaries are compressed using ZSTD

Kernel

Supporting more GPUs

- We are working with hardware vendors to expand the GPU support
- We are pushing towards SYCL and oneAPI via DPC++

Shading

Shading

- Two shading systems:
  - Cycles' Shader Virtual Machine (SVM), supported on all compute backends
  - OpenShadingLanguage (OSL), supported on CPU and OptiX
- Cycles uses shader network preprocessing
  - Unused nodes and dead branch removal
  - Constant folding
  - Node de-duplication

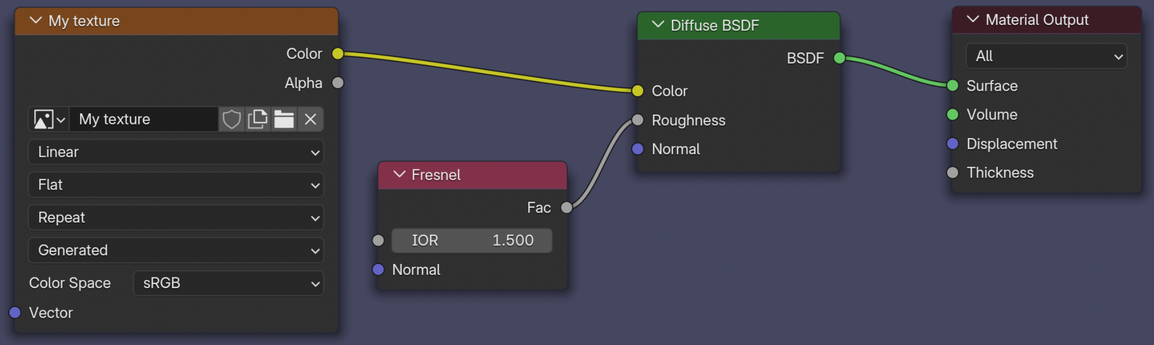
Shader Virtual Machine (SVM)

- Interactivity is the key principle of Cycles
- Specialized shaders could take a while to compile
- Instead, Cycles compiles shader trees to a special byte-code which is then interpreted in the shading kernel
  - No need to have a compiler! (SDK, LLVM JIT, etc.)
  - This byte-code of the SVM is represented as data
- The interpreter is the same for all shaders

Shader Virtual Machine (SVM)

- Shader graphs are compiled by Cycles into an array of uint4
- Nodes are compiled into one or multiple uint4
  - The first uint4 contains opcode, and could hold 3 uint parameters
  - Extra uint4 could be added to the SVM if node needs more data
- There is a stack from which node can read data, or to which node can write data
- The SVM in shading kernel interprets the array sequentially
  - Essentially, one big switch statement in a loop

Shading
SVM Compilation

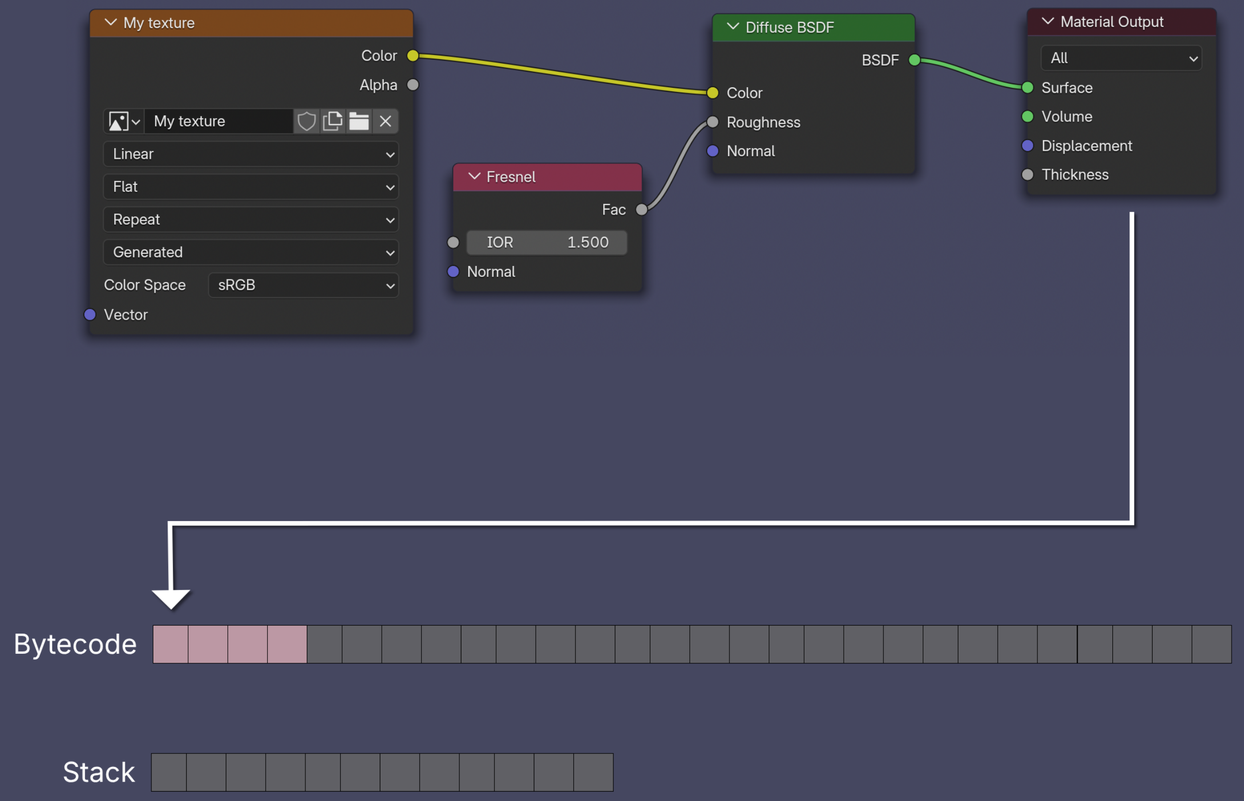


Bytecode

Stack

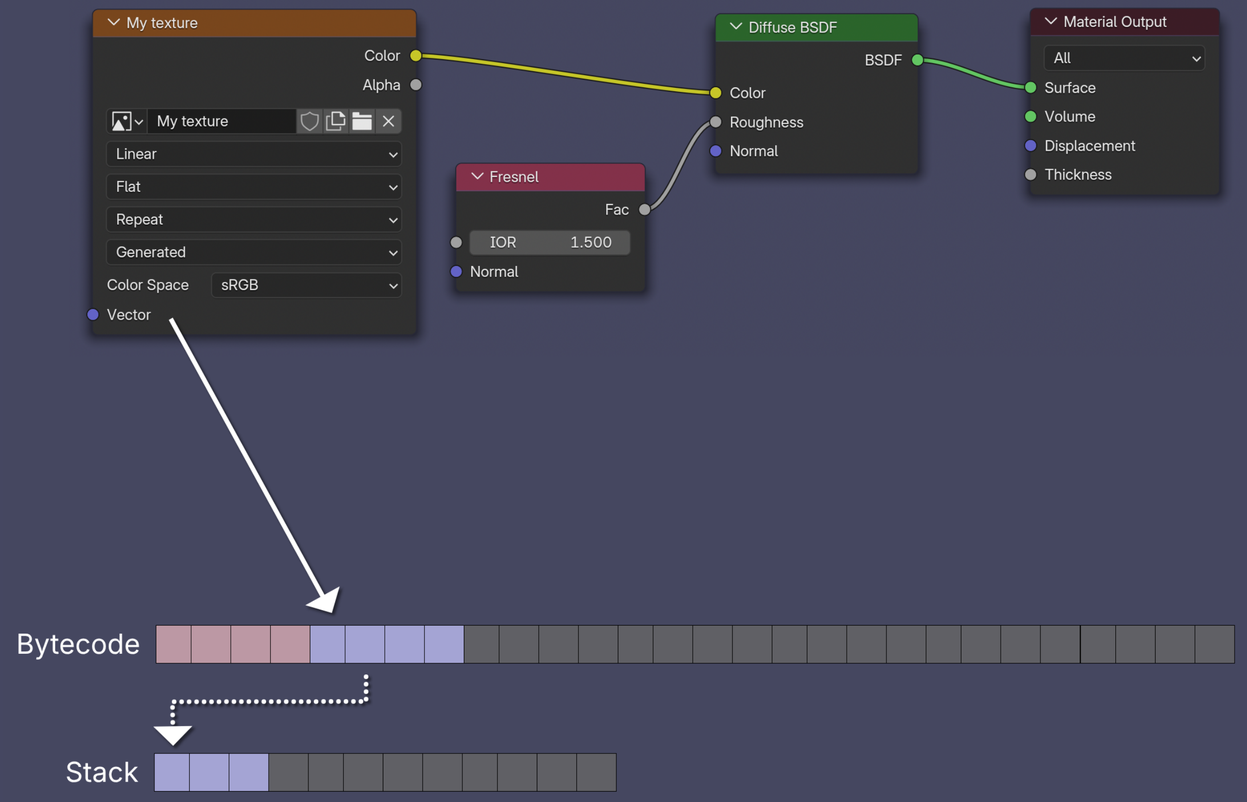
Shading
SVM Compilation

- Start with the Material Output
- Write byte-code:
  - Opcode NODE_SHADER_JUMP
  - Surface shader address: 0
  - Volume shader address: 0
  - Displacement shader address: 0
- Offsets are not yet known



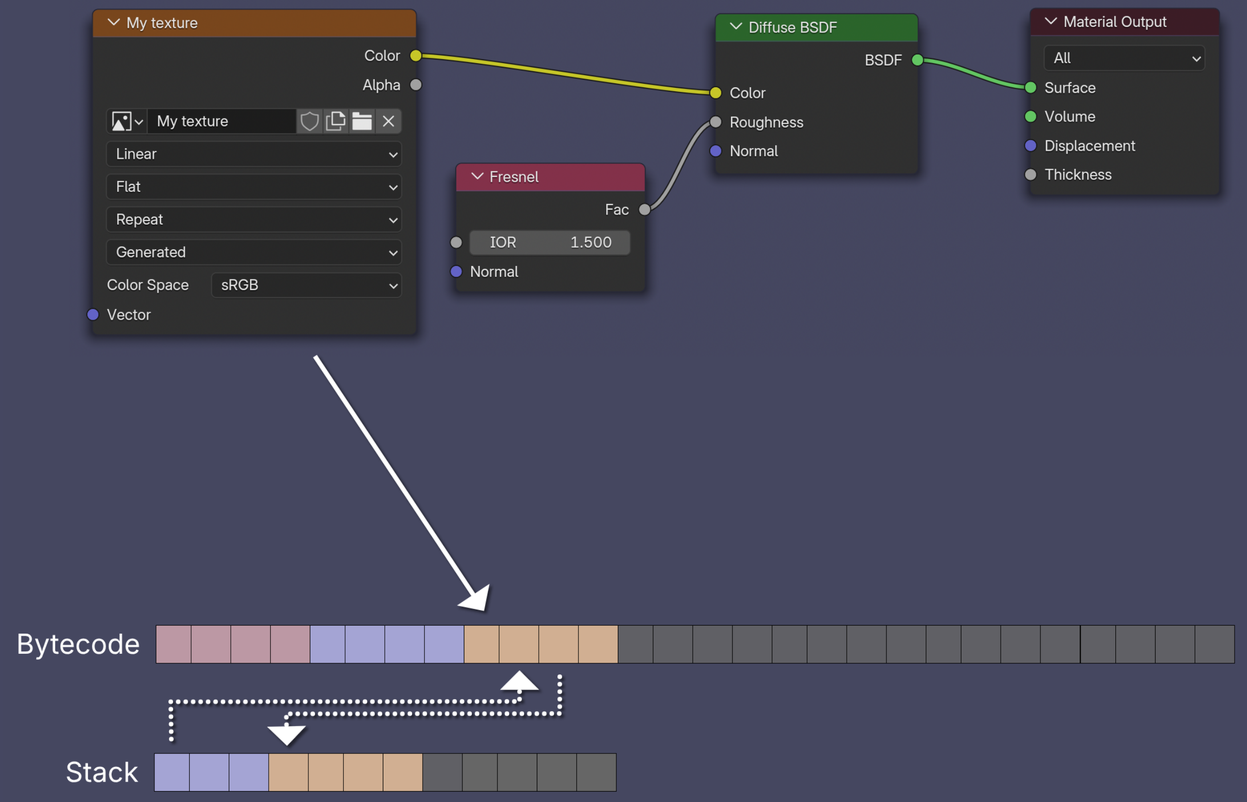
Shading
SVM Compilation

- Vector input of the Texture Node
  - Implicitly created as texture coordinate node
- Assign stack offset 0
  - This is where the result is written to
- Write byte-code:
  - Opcode: NODE_ATTR
  - Attribute type: ATTR_STD_UV
  - Output stack offset: 0



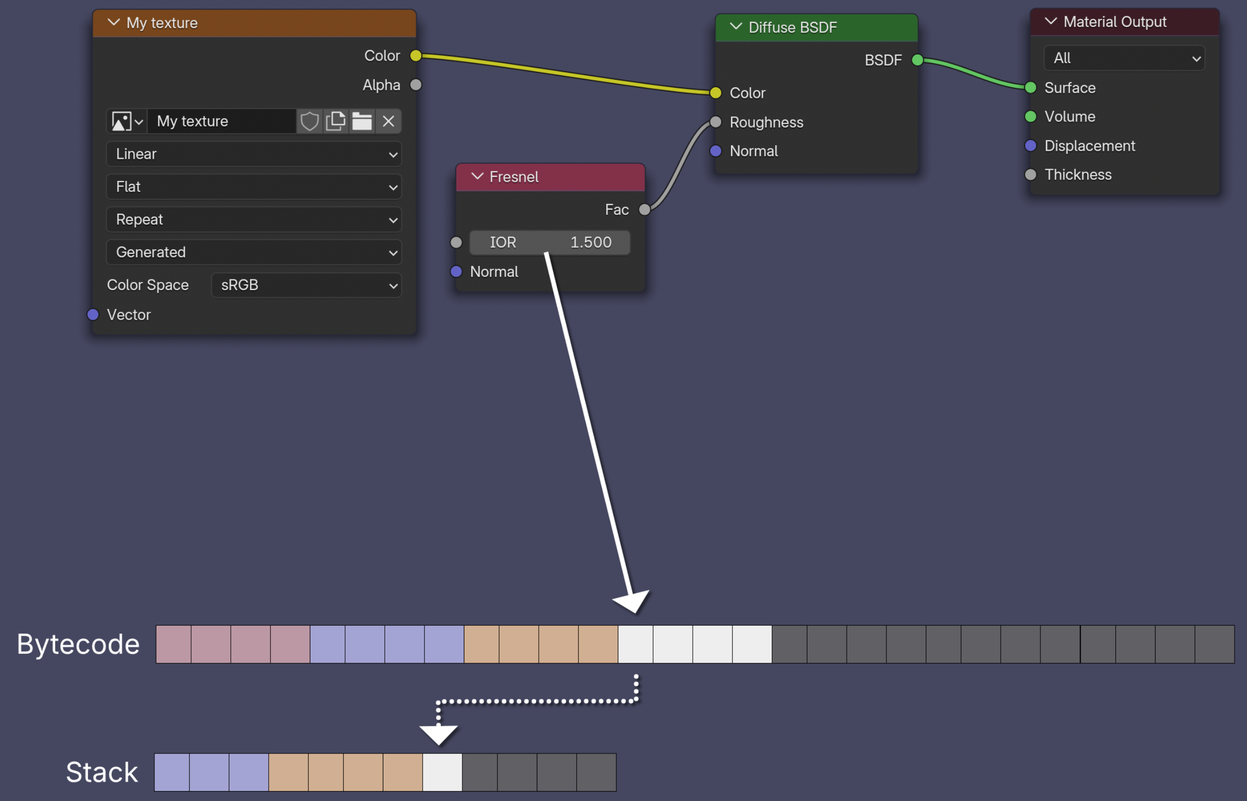
Shading
SVM Compilation

- Image Texture Node
- Assign stack offset 3
- Write byte-code:
  - Opcode: NODE_TEX_IMAGE
  - Coordinate stack offset: 0
  - Image slot: 0
  - Output stack address: 3
  - Other parameters like filtering



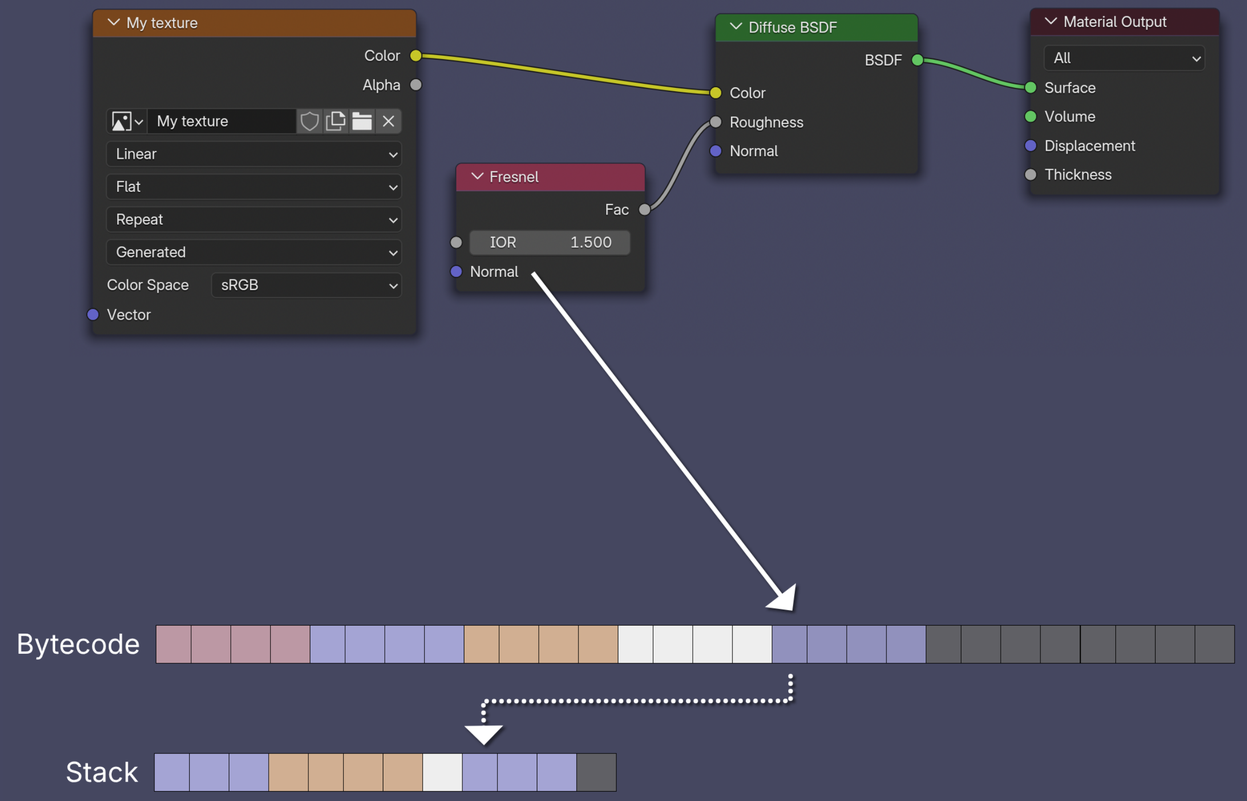
Shading
SVM Compilation

- Fresnel IOR
- Assign stack offset 7
- Write byte-code:
  - Opcode: NODE_VALUE
  - Output stack offset: 7



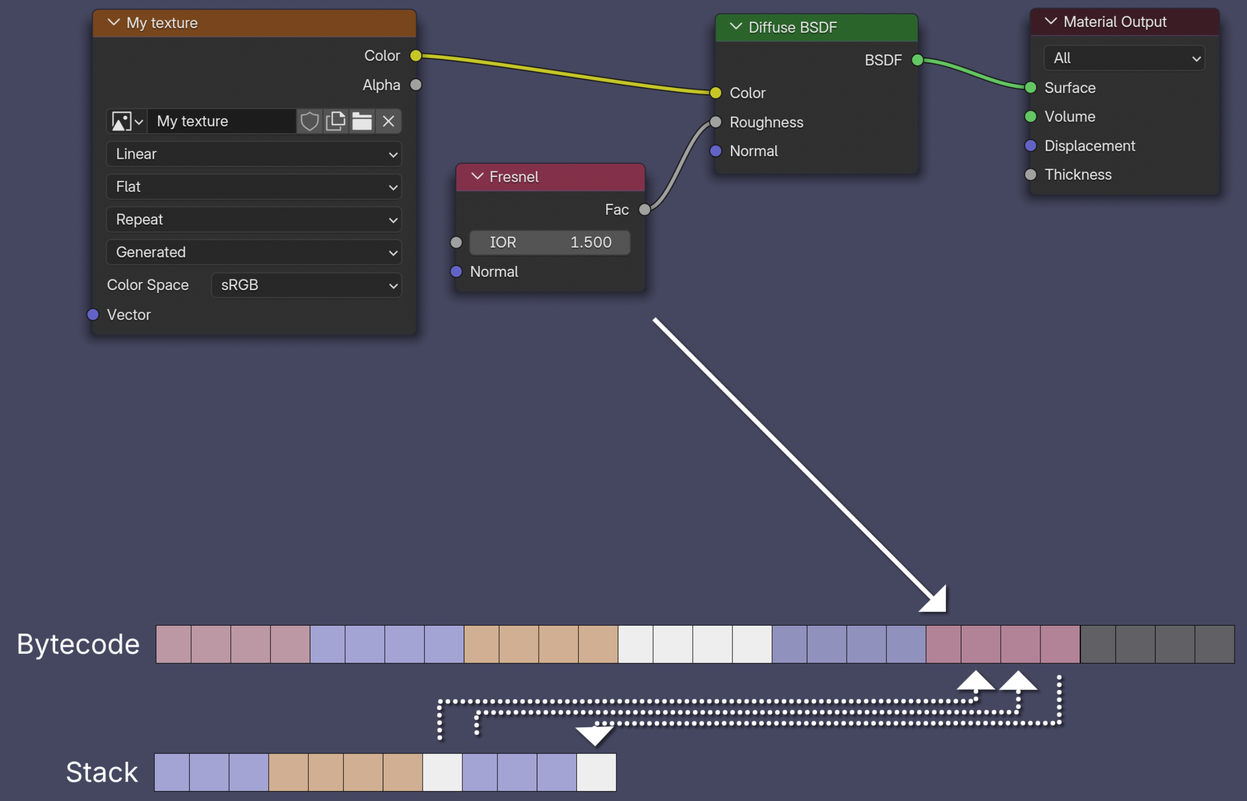
Shading
SVM Compilation

- Fresnel Normal
- Assign stack offset 8
- Write byte-code:
  - Opcode: NODE_GEOM_N
  - Output stack offset: 8



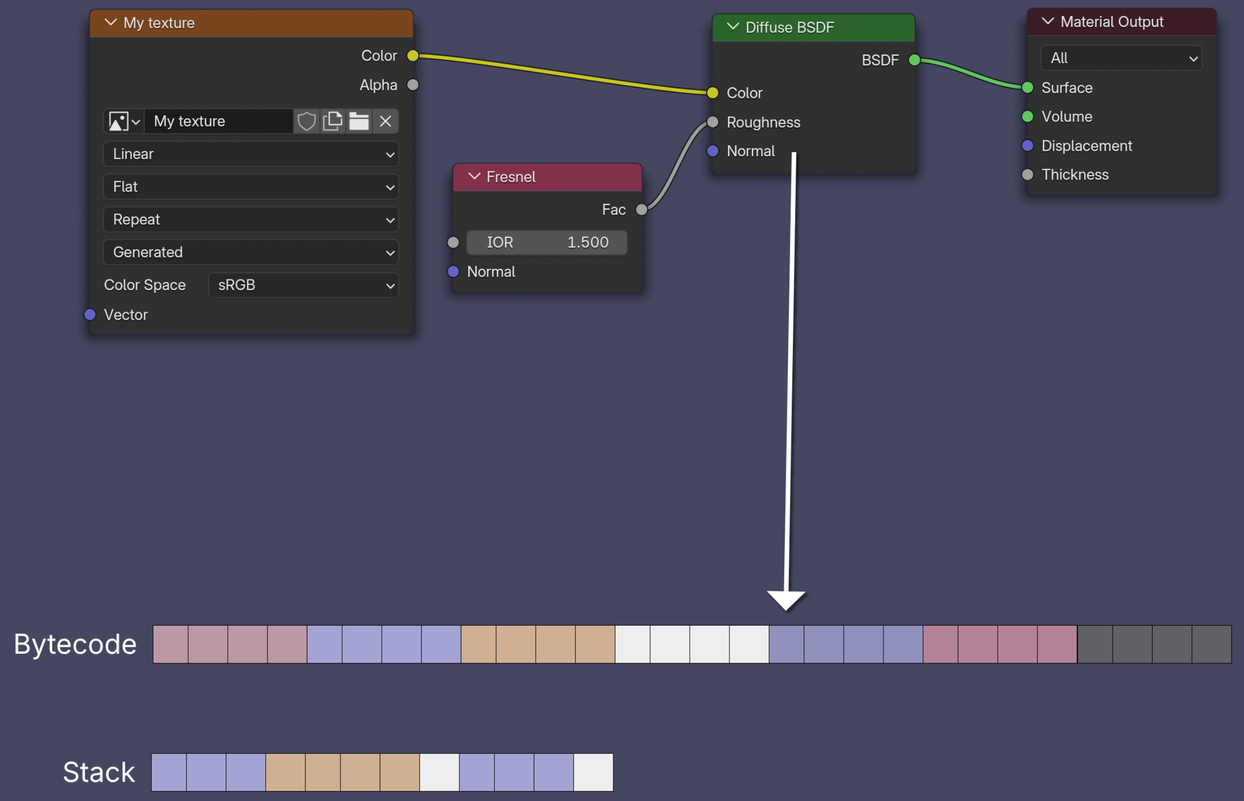
Shading
SVM Compilation

- Fresnel Node
- Assign stack offset 11
- Write byte-code:
  - Opcode: NODE_FRESNEL
  - IOR read offset: 3
  - Normal read offset: 4
  - Output stack offset: 11



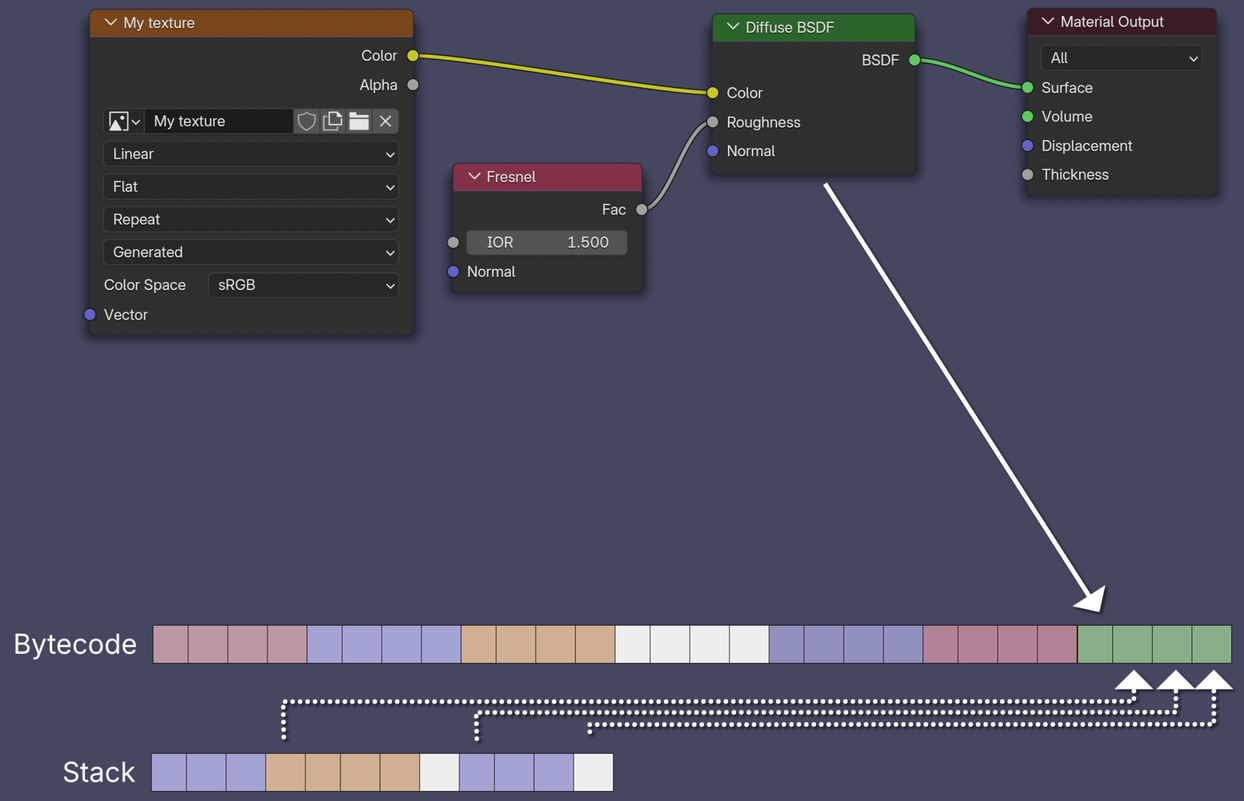
Shading
SVM Compilation

- Diffuse BSDF Normal
- Re-used normal from the Fresnel node evaluation



Shading
SVM Compilation

- Diffuse BSDF Node
- Write byte-code:
  - Opcode: NODE_CLOSURE_BSDF
  - Closure: Diffuse
  - Color read offset: 3
  - Normal read offset: 8
  - Roughness read offset: 11



Shader Virtual Machine (SVM)

Benefit:

- Minimal time until the first pixel
- Non-blocking interactive shader network edits

Downsides:

- Stresses GPU compilers A LOT
- Performance is lower than a dedicated shader
- Limited and fixed stack size

Shader Virtual Machine (SVM)

Future ideas

- Can SVM approach help interactivity of EEVEE?
- Will the same approach be used in Cycles for MaterialX?

Texture filtering

Texture filtering
Hardware filtering

- Whenever is possible hardware texture filtering is used
  - No hardware support on CPU
  - Compute backends provide accelerated linear texture sampling
  - Bicubic filtering on GPU is implemented using 4 bilinear lookups
- Adds a requirement to texture tiles to be padded
  - Note: the texture cache project is currently in a branch
- Volumes are handled via NanoVDB

Texture filtering

Stochastic filtering after shading

- Based on:

  Filtering After Shading With Stochastic Texture Filtering
  Matt Pharr, Bartlomiej Wronski, Marco Salvi, Marcos Fajardo

- Basic idea: utilize the Monte-Carlo integration and replace filter with a cheaper one:
  - Bilinear lookup can be replaced with nearest lookup
  - Bicubic lookup can be replaced with bilinear/nearest lookup
- Applicable on CPU and GPU, for 2D and 3D textures, but only implemented for 3D textures
- Only possible if shader only does linear operations with sampled texture value

Interactive viewport

Interactive viewport
General tricks

- Dynamic BVH
- Progressive resolution divider
- Denoising, on GPU when possible
- Currently looking into temporal denoising and upscaling
- Possible future development: limit the number of bounces during navigation

Interactive viewport
Statistics driven scheduling

- Cycles gather statistics
  - Path tracing and denoising time
  - GPU occupancy (threads utilization)
- Automatically calculates
  - Resolution divider used during updates
  - The number of samples to render between viewport updates
- Tries to maintain a decent refresh rate (currently 12fps)

References

- Megakernels Considered Harmful: Wavefront Path Tracing on GPUs
  Samuli Laine, et. al, NVIDIA
  https://research.nvidia.com/sites/default/files/pubs/2013-07_Megakernels-Considered-Harmful/laine2013hpg_paper.pdf

- The Iray Light Transport Simulation and Rendering System
  A. Keller, et. al, NVIDIA
  https://raytracing-docs.nvidia.com/iray/presentations/iray_overview/nvidia_iray_rendering_system.pdf

- Filtering After Shading With Stochastic Texture Filtering
  Matt Pharr, Bartlomiej Wronski, Marco Salvi, Marcos Fajardo
  https://d1qx31qr3h6wln.cloudfront.net/publications/stochtex.pdf

Thank you!