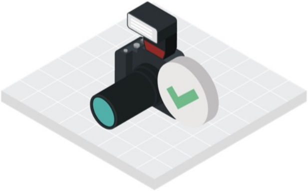
thrive
SIGGRAPH2019
LOS ANGELES • 28 JULY – 1 AUGUST

RODENT: GENERATING RENDERERS WITHOUT WRITING A GENERATOR

A. Pérard-Gayot, R. Membarth, R. Leissa, S. Hack, P. Slusallek

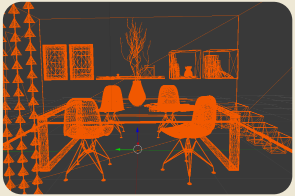
UNIVERSITÄT DES SAARLANDES

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH



PHOTOGRAPHY & RECORDING ENCOURAGED

Overview



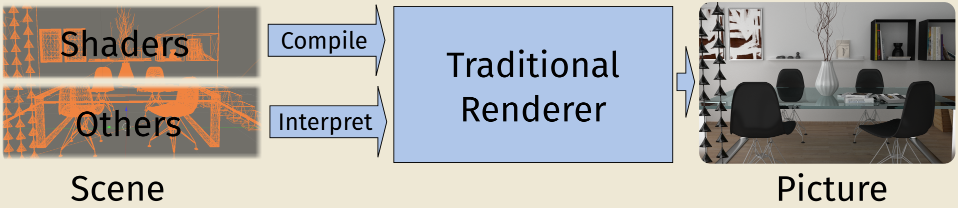
Scene

Traditional Renderer

Picture

What this talk is about

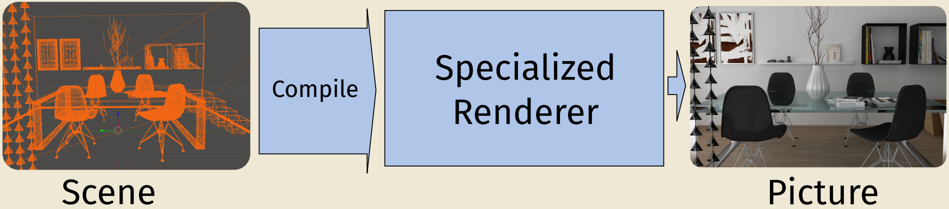
- Generating renderers from high-level, textbook-like code
- Specialized/optimized for a scene type
- High-performance: Up to 40%/20% faster than OptiX/Embree+`ispc`



In a traditional renderer

- Shaders are compiled by a (shader) compiler
  - Standard compiler optimizations
- Rest of the scene is interpreted during rendering
  - `if`/`else` branches (e.g. for renderer config/options)
  - Virtual function calls (e.g. for geometry types)
  - …

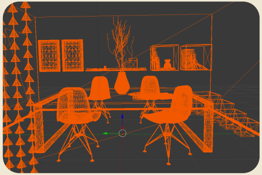
Rendering



In Rodent

- We compile the entire scene into a renderer
- We only use the scene type, not the actual scene data
    - No benefit from knowing e.g. the position of triangle 544
- We use Partial Evaluation
    - To avoid writing a *Renderer Generator*

Traditional Execution vs. Partial Evaluation



Scene

Traditional Renderer

Picture

Traditional program execution

Traditional Execution vs. Partial Evaluation

High-level
Rendering
Code

Partial Evaluation

Traditional Execution vs. Partial Evaluation

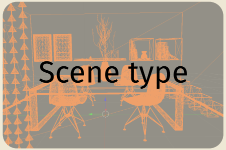
High-level
Rendering
Code

Rodent

Traditional Execution vs. Partial Evaluation

High-level
Rendering
Code

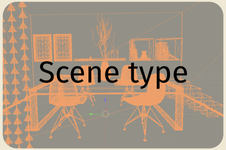
Rodent



Traditional Execution vs. Partial Evaluation

High-level Rendering Code

Rodent

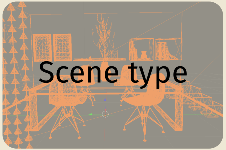


Partial Evaluator

Traditional Execution vs. Partial Evaluation

High-level Rendering Code

Rodent

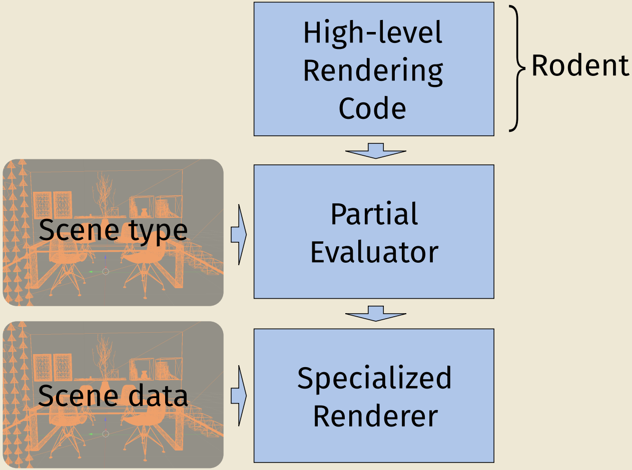


Partial Evaluator

Specialized Renderer

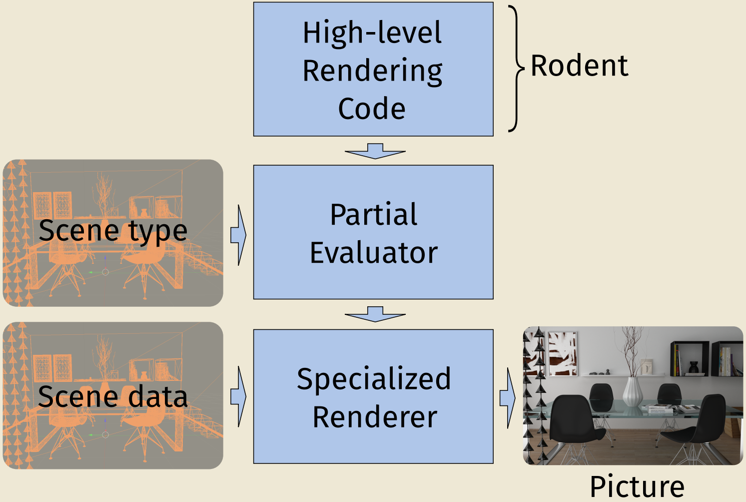
Partial Evaluation

Traditional Execution vs. Partial Evaluation



Partial Evaluation

Traditional Execution vs. Partial Evaluation



Partial Evaluation

AnyDSL

- This work leverages the AnyDSL compiler framework
    - `https://github.com/AnyDSL`
- Provides user-guided Partial Evaluation
- High-performance code generation using LLVM
- Can target/optimize for CPUs or GPUs
    - Intel/AMD/NVIDIA/ARM/…

Rendering Library Design

- High-level, textbook-like
  - In the spirit of PBRT
- Descriptive and modular
  - Separate the algorithm (”what”) from the schedule/hardware mapping (”how”)
- High-performance
  - Different hardware *mappings*
  - CPUs/GPUs have different execution models
  - Need efficient and flexible abstractions

The ”What”

BSDF

```
struct Bsdf {
  // Evaluation of the function given a pair of directions
  eval: fn (Vec3, Vec3) -> Color,

  // Probability density function used during sampling
  pdf: fn (Vec3, Vec3) -> f32,

  // Samples a direction (importance sampled according to this BSDF)
  sample: fn (Vec3) -> BsdfSample,
}
```

Example: Diffuse BSDF

```
fn @make_diffuse_bsdf(surf: SurfaceElement, kd: Color) -> Bsdf {
  Bsdf {
    eval: @ |in_dir, out_dir| kd * (1.0f / pi),
    pdf:  @|in_dir, out_dir|
      cosine_hemisphere_pdf(positive_cos(in_dir, surf.normal)),
    sample: @ |out_dir| {
      let sample = sample_cosine_hemisphere(rand(), rand());
      let color  = kd * (1.0f / pi);
      make_bsdf_sample(surf, sample, color)
    }
  }
}
```

- @ triggers partial evaluation/specializes the function
- Replaces the function by its contents at the call site to allow optimizations

Defining a scene with Rodent

- BSDFs:

```
let diff = make_diffuse_bsdf(kd);
let spec = make_phong_bsdf(ns, ks);
let bsdf = make_mix_bsdf(spec, diff, k);
```

Defining a scene with Rodent

- BSDFs:

```
let diff = make_diffuse_bsdf(kd);
let spec = make_phong_bsdf(ns, ks);
let bsdf = make_mix_bsdf(spec, diff, k);
```

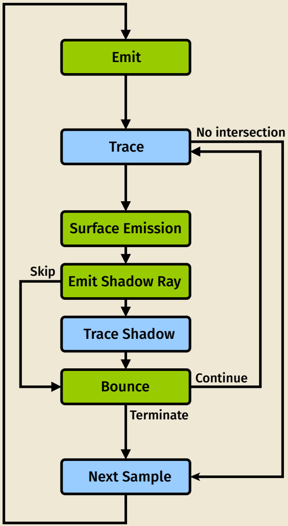
- Light sources, textures, geometric objects, ...

Rodent is a Scene Description Language

```
let renderer = make_path_tracing_renderer(/* ... */);
let geometry = make_tri_mesh_geometry(/* ... */);

let tex = make_image_texture(/* ... */);

let shader = |ray, hit, surface| {
    let uv = surface.attribute(0).as_vec2;
    make_diffuse_bsdf(surface, tex(uv1));
};

let scene = make_scene(geometry, /* ... */);
```

BSDF DSL + Light DSL + Geometry DSL + … = Scene language embedded in AnyDSL

Abstracting the Rendering Process




```
struct Tracer {
  on_emit:   OnEmitFn,
  on_hit:    OnHitFn,
  on_shadow: OnShadowFn,
  on_bounce: OnBounceFn,
}
```

- Can also be used for bidir. algorithms
- Green nodes: the algorithm
  What should be computed
- Blue nodes: the schedule
  How it should be computed

The ”How”

Mapping Renderers to Hardware

- The `Device` contains hardware-specific routines:

```
struct Device {
  trace: fn (Scene, Tracer) -> (),
  /* ... */
}
```

- Schedule renderers differently depending on the platform
    - Wavefront: Batches (larger than SIMD width) of rays together
    - Megakernel: Large compute kernel, one ray at a time (used in OptiX)
- Rodent implements 3 devices:
    1. CPU: Wavefront
    2. GPU: Megakernel
    3. GPU: Wavefront

Wavefront Devices

On CPUs

- Processes a small (∼1000 rays) batch of rays together
  - Maximize cache efficiency
- Sort rays by shader and process contiguous ranges
- Uses vectorization and specialization, simplified:

```
for shader in unroll(0, scene.num_shaders) {
  // Get the range of rays for this shader
  let (begin, end) = ray_range_by_shader(shader);
  for i in vectorize(vector_width, begin, end) {
    // Scalar code using on_hit(), on_shadow(), ...
    // => automatically vectorized
  }
}
```

Wavefront Devices

On CPUs

- Processes a small (∼1000 rays) batch of rays together
  - Maximize cache efficiency
- Sort rays by shader and process contiguous ranges
- Uses vectorization and specialization, simplified:

```
for shader in unroll(0, scene.num_shaders) {
  // Get the range of rays for this shader
  let (begin, end) = ray_range_by_shader(shader);
  for i in vectorize(vector_width, begin, end) {
    // Scalar code using on_hit(), on_shadow(), ...
    // => automatically vectorized
  }
}
```

```
i∈unroll(0,3)
 ┗j∈vectorize(w,begin(i),end(i))
            ⬇
j0∈vectorize(w,begin(0),end(0))
j1∈vectorize(w,begin(1),end(1))
j2∈vectorize(w,begin(2),end(2))
```

Wavefront Devices

On GPUs

- Processes a larger (∼1M rays) batch of rays
  - Maximize parallelism
- Sort rays by shader and process contiguous ranges
- Generates one kernel per shader, with specialization, simplified:

```
for shader in unroll(0, scene.num_shaders) {
  // Get the range of rays for this shader
  let (begin, end) = ray_range_by_shader(shader);
  let grid = (round_up(end - begin, block_size), 1, 1);
  let block = (block_size, 1, 1);
  with work_item in cuda(grid, block) {
    // Use on_hit(), on_shadow(), ...
  }
}
```

Wavefront Devices

On GPUs

- Processes a larger (∼1M rays) batch of rays
  - Maximize parallelism
- Sort rays by shader and process contiguous ranges
- Generates one kernel per shader, with specialization, simplified:

```
for shader in unroll(0, scene.num_shaders) {
  // Get the range of rays for this shader
  let (begin, end) = ray_range_by_shader(shader);
  let grid = (round_up(end - begin, block_size), 1, 1);
  let block = (block_size, 1, 1);
  with work_item in cuda(grid, block) {
    // Use on_hit(), on_shadow(), ...
  }
}
```

```
i∈unroll(0,3)
 └cuda(grid(i),block(i))
          ⬇
cuda(grid(0),block(0))
cuda(grid(1),block(1))
cuda(grid(2),block(2))
```

Megakernel GPU Device

- Rays are local to the current execution thread
- Rendering loop *inside* the kernel, simplified:

```
fn trace(scene: Scene, tracer: Tracer) -> () {
  with work_item in cuda(grid, block) {
    let (x, y) = (work_item.gidx(), work_item.gidy());
    let (ray, state) = tracer.on_emit(x, y);
    let mut terminated = false;
    while !terminated {
      // Trace + use on_hit(), on_shadow(), ...
    }
  }
}
```

Evaluation

- Versus high-performance, state-of-the-art frameworks:
  - Embree + `ispc`: only for x86/amd64
  - OptiX: only for CUDA hardware
- Built custom, simple renderers based on those frameworks
  - Following documentation
  - Only implemented features required to render the test scenes
- Measured:
  - Performance
  - Code complexity
- Workflow: Convert scene to AnyDSL $\Rightarrow$ compile $\Rightarrow$ render

Scenes

786k tris./ 13 mats.

1.231M tris./14 mats.

545k tris./35 mats.

718k tris./44 mats.

612k tris./61 mats.

263k tris./23 mats.

Results: Performance

	CPU (Intel™ i7 6700K)		GPU (NVIDIA™ Titan X)			GPU (AMD™ R9 Nano)	
Scene	Rodent[2]	Embree	Rodent[1]	Rodent[2]	OptiX	Rodent[1]	Rodent[2]
Living Room	9.77 (+23%)	7.94	38.59 (+25%)	43.52 (+42%)	30.75	24.87	35.11
Bathroom	6.65 (+13%)	5.90	27.06 (+31%)	35.32 (+42%)	20.64	14.95	27.31
Bedroom	7.55 (+ 4%)	7.24	30.25 (+ 9%)	38.88 (+29%)	27.72	19.25	32.90
Dining Room	7.08 (+ 1%)	7.01	30.07 (+ 5%)	40.37 (+29%)	28.58	16.22	30.83
Kitchen	6.64 (+12%)	5.92	22.73 (+ 2%)	32.09 (+31%)	22.22	16.68	28.13
Staircase	4.86 (+ 8%)	4.48	20.00 (+18%)	27.53 (+39%)	16.89	11.74	22.21

(1) Megakernel, (2) Wavefront

Results: Performance

	CPU (Intel™ i7 6700K)		GPU (NVIDIA™ Titan X)			GPU (AMD™ R9 Nano)	
Scene	Rodent[2]	Embree	Rodent[1]	Rodent[2]	OptiX	Rodent[1]	Rodent[2]
Living Room	9.77 (+23%)	7.94	38.59 (+25%)	43.52 (+42%)	30.75	24.87	35.11
Bathroom	6.65 (+13%)	5.90	27.06 (+31%)	35.32 (+42%)	20.64	14.95	27.31
Bedroom	7.55 (+ 4%)	7.24	30.25 (+ 9%)	38.88 (+29%)	27.72	19.25	32.90
Dining Room	7.08 (+ 1%)	7.01	30.07 (+ 5%)	40.37 (+29%)	28.58	16.22	30.83
Kitchen	6.64 (+12%)	5.92	22.73 (+ 2%)	32.09 (+31%)	22.22	16.68	28.13
Staircase	4.86 (+ 8%)	4.48	20.00 (+18%)	27.53 (+39%)	16.89	11.74	22.21

(1) Megakernel, (2) Wavefront

- Between $+1 - 23\%$ vs. Embree
    - Around $60 - 70\%$ of the time tracing rays
    - Traversal algorithms in Embree are already specialized
    - Rodent's shading alone is around $2\times$ faster than with `ispc`

Results: Performance

	CPU (Intel™ i7 6700K)		GPU (NVIDIA™ Titan X)			GPU (AMD™ R9 Nano)	
Scene	Rodent[2]	Embree	Rodent[1]	Rodent[2]	OptiX	Rodent[1]	Rodent[2]
Living Room	9.77 (+23%)	7.94	38.59 (+25%)	43.52 (+42%)	30.75	24.87	35.11
Bathroom	6.65 (+13%)	5.90	27.06 (+31%)	35.32 (+42%)	20.64	14.95	27.31
Bedroom	7.55 (+ 4%)	7.24	30.25 (+ 9%)	38.88 (+29%)	27.72	19.25	32.90
Dining Room	7.08 (+ 1%)	7.01	30.07 (+ 5%)	40.37 (+29%)	28.58	16.22	30.83
Kitchen	6.64 (+12%)	5.92	22.73 (+ 2%)	32.09 (+31%)	22.22	16.68	28.13
Staircase	4.86 (+ 8%)	4.48	20.00 (+18%)	27.53 (+39%)	16.89	11.74	22.21

(1) Megakernel, (2) Wavefront

- Between $+1 - 23\%$ vs. Embree
  - Around $60 - 70\%$ of the time tracing rays
  - Traversal algorithms in Embree are already specialized
  - Rodent's shading alone is around $2\times$ faster than with `ispc`
- Between $+2 - 31\%$ vs OptiX (Megakernel)

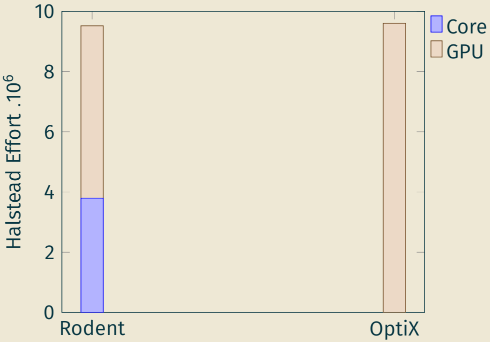
Results: Performance

	CPU (Intel™ i7 6700K)		GPU (NVIDIA™ Titan X)			GPU (AMD™ R9 Nano)	
Scene	Rodent[2]	Embree	Rodent[1]	Rodent[2]	OptiX	Rodent[1]	Rodent[2]
Living Room	9.77 (+23%)	7.94	38.59 (+25%)	43.52 (+42%)	30.75	24.87	35.11
Bathroom	6.65 (+13%)	5.90	27.06 (+31%)	35.32 (+42%)	20.64	14.95	27.31
Bedroom	7.55 (+ 4%)	7.24	30.25 (+ 9%)	38.88 (+29%)	27.72	19.25	32.90
Dining Room	7.08 (+ 1%)	7.01	30.07 (+ 5%)	40.37 (+29%)	28.58	16.22	30.83
Kitchen	6.64 (+12%)	5.92	22.73 (+ 2%)	32.09 (+31%)	22.22	16.68	28.13
Staircase	4.86 (+ 8%)	4.48	20.00 (+18%)	27.53 (+39%)	16.89	11.74	22.21

(1) Megakernel, (2) Wavefront

- Between $+1 - 23\%$ vs. Embree
  - Around $60 - 70\%$ of the time tracing rays
  - Traversal algorithms in Embree are already specialized
  - Rodent's shading alone is around $2\times$ faster than with `ispc`
- Between $+2 - 31\%$ vs OptiX (Megakernel)
- Between $+29 - 42\%$ vs OptiX (Wavefront)
  - Wavefront scales better with shader complexity
  - Not limited by register pressure

Results: Code Complexity



- Embree: only on x86/amd64
- Rodent: also on ARM
  + other LLVM targets (RISC-V?)

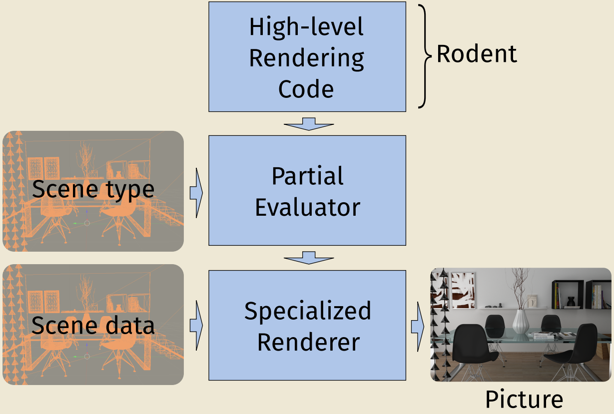
- OptiX: only Megakernel, only CUDA hw.
- Rodent: also on AMD™ GPUs
  + other LLVM targets (Intel™ GPU?)

Conclusion

Rodent *generates* high-performance renderers *without writing a generator*

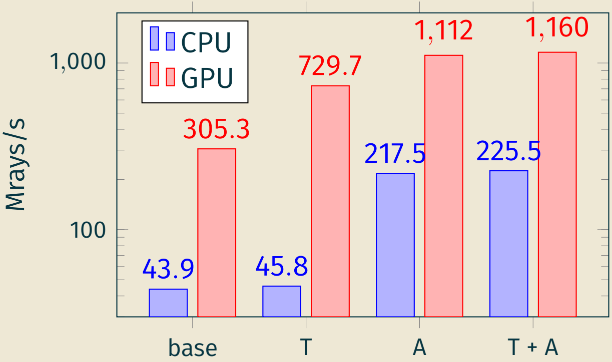
- Defines textbook-like, generic algorithms
- Provides tailored hardware schedules for different CPUs and GPUs
- Specializes code according to the scene via AnyDSL
- Runs up to 40% faster than state-of-the-art

Questions?



https://github.com/AnyDSL/rodent

Results: Impact of Specialization

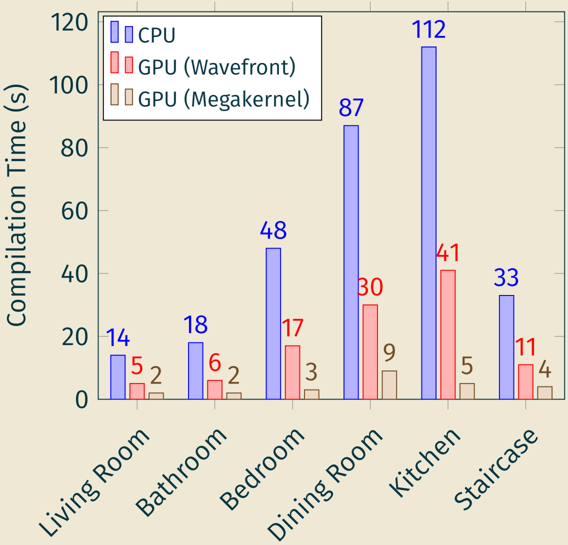


- Base: No specialization
- T: Specialize the interface (shader $\longleftrightarrow$ texturing function)
- A: Specialize the interface (shader $\longleftrightarrow$ mesh attribute)

Specialization: Caveats

- Specialization may lead to increased compilation times
- Specializing to much may increase register pressure
  - Dangerous for the megakernel device
  - Not a problem for the wavefront device
- Rodent fuses simple/similar shaders together
  - Only for the megarkernel device
  - Mitigates problems of divergence and reg. pressure

Results: Compilation Times



Improving Compilation Times

- The more there is to specialize, the slower
- Compiler itself is not particularly optimized for speed
- Parts of the renderer can be pre-compiled
- Does not need to know *everything* in the scene
    - The less is known the less specialization will happen
    - Automatically done by the compiler thanks to annotations
    - Can be exploited to make compilation faster