DiffTaichi

DIFFERENTIABLE PROGRAMMING FOR PHYSICAL SIMULATION

Authors:

Yuanming Hu, Luke Anderson, Tzu-Mao Li, Qi Sun, Nathan Carr, Jonathan Ragan-Kelley, Frédo Durand

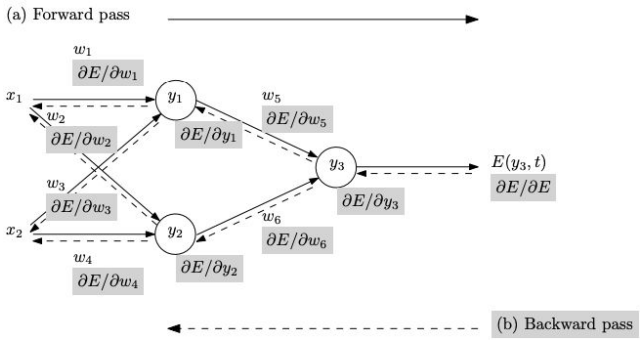
Presenter:

Abhay Pratap Singh Rathore, 2019CS50414

Contents

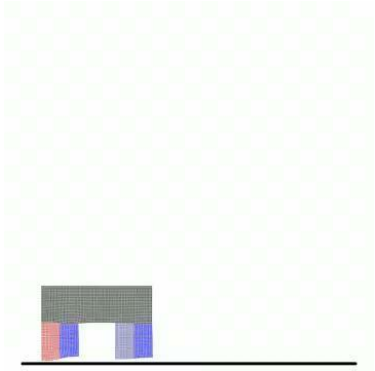
Videos Link:
https://drive.google.com/drive/folders/151aKNekE43sco32Mi3uf1BU3mVgl9kTx?usp=sharing

What is DiffTaichi?



Differentiable Programming Language

Physical Simulation



```python
@ti.func
def complex_sqr(z):  # complex square of a 2D vector
    return tm.vec2(z[0] * z[0] - z[1] * z[1], 2 * z[0] * z[1])

@ti.kernel
def paint(t: float):
    for i, j in pixels:  # Parallelized over all pixels
        c = tm.vec2(-0.8, tm.cos(t) * 0.2)
        z = tm.vec2(i / n - 1, j / n - 0.5) * 2
        iterations = 0
        while z.norm() < 20 and iterations < 50:
            z = complex_sqr(z) + c
            iterations += 1
        pixels[i, j] = 1 - iterations * 0.02
```

Source Code Transformations

Embedded in Python 🙂

Approach	Forward Time	Backward Time	Total Time	# Lines of Code
TensorFlow	13.20 ms	35.70 ms	48.90 ms (188.×)	190
CUDA	0.10 ms	0.14 ms	0.24 ms (0.92×)	460
DiffTaichi	0.11 ms	0.15 ms	0.26 ms (1.00×)	110

Example Code: Mass-Spring System

```
@ti.func
def apply_spring_force(t: ti.int32):
    for i in range(num_springs):
        a = spring_anchor_a[i]
        b = spring_anchor_b[i]
        spring_vec = x[t, a] - x[t, b]
        spring_force = -spring_stiffness[i] * (
            spring_vec.norm() - spring_length[i]
        ) * spring_vec.normalized()
        f[t, a] += spring_force
        f[t, b] -= spring_force
```

```
@ti.func
def integrate(t: ti.int32):
    for i in range(num_particles):
        v[t + 1, i] = v[t, i] + dt * f[t, i]
        x[t + 1, i] = (
            x[t, i] +
            0.5 * dt * (v[t, i] + v[t + 1, i])
        )
```

```
# Assembling the Forward Simulator

def forward():
   for t in range(1, steps):
     apply_spring_force(t)
     integrate(t)
```

```
# Global variables

x = ti.Vector.field(2, dtype=real, shape=(num_steps, num_particles))
v = ti.Vector.field(2, dtype=real, shape=(num_steps, num_particles))
f = ti.Vector.field(2, dtype=real, shape=(num_steps, num_particles))

spring_anchor_a = ti.field(dtype=ti.i32, shape=num_springs)
spring_anchor_b = ti.field(dtype=ti.i32, shape=num_springs)
spring_length = ti.field(dtype=real, shape=num_springs)
spring_stiffness = ti.field(dtype=real, shape=num_springs)
```

DiffTaichi automatic differentiation

Imperative Programming Language

1. Allows if-else statements (Important in many physics simulation, like collision detection).
2. Easier to port existing physical simulation code to DiffTaichi.

Flexible Indexing

1. Existing parallel differentiable programming systems provide element-wise operations on arrays of the same shape. c[i, j] = a[i, j] + b[i, j].
2. Allows updating arrays via arbitrary indexing. Necessary in many physical simulations. y[p[i] * 2, j] = x[q[i + j]].

Megakernels

1. Uses source code transformations and just-in-time compilers.
2. Not present in TensorFlow and PyTorch.

```
@ti.kernel
def advect(field: ti.template(), field_out: ti.template(),
           t_offset: ti.template(), t: ti.i32):
    """Move field smoke according to x and y velocities (vx and vy)
     using an implicit Euler integrator."""
    for y in range(n_grid):
        for x in range(n_grid):
            center_x = y - v[t + t_offset, y, x][0]
            center_y = x - v[t + t_offset, y, x][1]

            # Compute indices of source cell
            left_ix = ti.cast(ti.floor(center_x), ti.i32)
            top_ix = ti.cast(ti.floor(center_y), ti.i32)

            rw = center_x - left_ix  # Relative weight of right-hand cell
            bw = center_y - top_ix  # Relative weight of bottom cell

            # Wrap around edges
            # TODO: implement mod (%) operator
            left_ix = imod(left_ix, n_grid)
            right_ix = inc_index(left_ix)
            top_ix = imod(top_ix, n_grid)
            bot_ix = inc_index(top_ix)

            # Linearly-weighted sum of the 4 surrounding cells
            field_out[t, y, x] = (1 - rw) * (
                (1 - bw) * field[t - 1, left_ix, top_ix] +
                bw * field[t - 1, left_ix, bot_ix]) + rw * (
                    (1 - bw) * field[t - 1, right_ix, top_ix] +
                    bw * field[t - 1, right_ix, bot_ix])
```

DiffTaichi AD: Two Scale AD

Source Code Transformation

High Performance

Poor Flexibility

Long compilation time

Tracing

Flexibility

Poor Performance

Two-Scale Automatic Differentiation System

1. SCT for differentiating within kernels.
   High Performance
2. Light-weight tape that only stores function pointers and arguments for end-to-end simulation differentiation.
   Flexibility

Forward Program

```
with ti.Tape(loss)
  for i in range(3):
    compute_force(i)
    move_partcies(i)
  compute_loss()
```

Tape Contents

```
compute_force, args=(0)
move_partcies, args=(0)
compute_force, args=(1)
move_partcies, args=(1)
compute_force, args=(2)
move_partcies, args=(2)
compute_loss, args=()
```

Backward Program

```
compute_loss.grad()
move_partcies.grad(2)
compute_force.grad(2)
move_partcies.grad(1)
compute_force.grad(1)
move_partcies.grad(0)
compute_force.grad(0)
```

DiffTaichi AD: Assumptions

Global Data Access Rules

1. If a global tensor element is written more than once, then starting from the second write, the write must come in the form of an atomic add ("accumulation").
2. No read accesses happen to a global tensor element, until its accumulation is done.

Avoid mixed usage of parallel for-loop and non-for statements

```
@ti.kernel
def differentiable_task():
    loss[None] += x[0]
    for i in range(10):
        ...
```

(primal) $f(X_0, X_1, .., X_n) = Y_0, Y_1, \ldots, Y_m$

↓ Reverse-Mode Automatic Differentiation

(adjoint) $f^*(X_0, X_1, .., X_n, Y_0^*, Y_1^*, \ldots, Y_m^*) = X_0^*, X_1^*, .., X_n^*.$

Rules can be validated

ti.init(debug=True)
ti.ad.Tape(..., validation=True)

Local AD: Source Code Transformation

Preprocessing

1. Flatten Branching
2. Eliminate Mutable Local Variables

Loops

1. Keeps parallel loops as it is
2. Reverse the order of non-parallel loops in differentiation.
3. No local mutable variables allowed. to avoid complex history tracking.
4. Use global variables instead.

```
int a = 0;
if (b > 0) { a = b;}
  else     { a = 2b;}
a = a + 1;
return a;
```

```
// flatten branching
int a = 0;
a = select(b > 0, b, 2b);
a = a + 1
return a;
```

```
// eliminate mutable var

ssa1 = select(b > 0, b, 2b);
ssa2 = ssa1 + 1
return ssa2;
```

Local AD: Source Code Transformation

```
y[i] = sin(x[i] * x[i])
```

```
for i ∈ range(0, 16, step 1) do
    %1 = load x[i]
    %2 = mul %1, %1
    %3 = sin(%2)
    store y[i] = %3
end for
```

Primal Kernel

```
for i in range(0, 16, step 1) do
    // adjoint variables
    %1adj = alloca 0.0
    %2adj = alloca 0.0
    %3adj = alloca 0.0
    // original forward computation
    %1 = load x[i]
    %2 = mul %1, %1
    %3 = sin(%2)
    // reverse accumulation
    %4 = load y_adj[i]
    %3adj += %4
    %5 = cos(%2)
    %2adj += %3adj * %5
    %1adj += 2 * %1 * %2adj
    atomic add x_adj[i], %1adj
end for
```

Adjoint Kernel

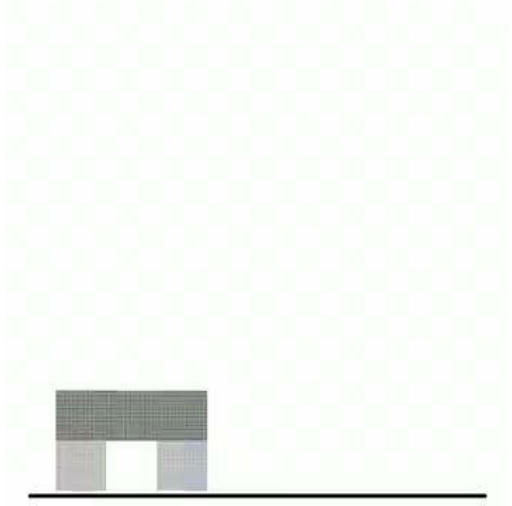
Global AD: Using a Light-Weight Tape

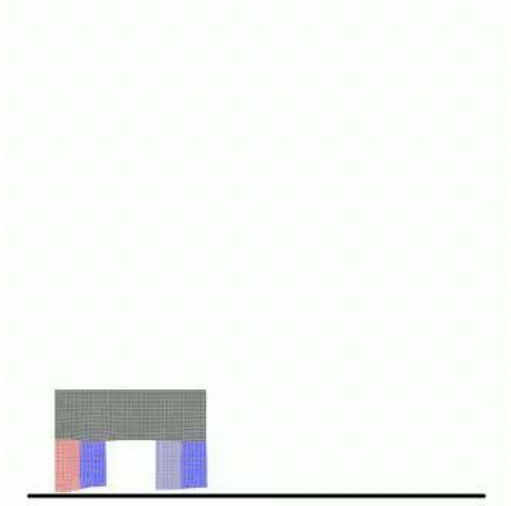
Light weight: Only stores points and scalar input params.

Forward Program	Tape Contents	Backward Program

`with ti.Tape(loss)`
`  for i in range(3):`
`  compute_force(i)`
`  move_partcies(i)`
`  compute_loss()`
	`compute_force, args=(0)`
	`move_partcies, args=(0)`
	`compute_force, args=(1)`
	`move_partcies, args=(1)`
	`compute_force, args=(2)`
	`move_partcies, args=(2)`
	`compute_loss, args=()`
		`compute_loss.grad()`
		`move_partcies.grad(2)`
		`compute_force.grad(2)`
		`move_partcies.grad(1)`
		`compute_force.grad(1)`
		`move_partcies.grad(0)`
		`compute_force.grad(0)`

Evaluation





Approach	Forward Time	Backward Time	Total Time	# Lines of Code
TensorFlow	13.20 ms	35.70 ms	48.90 ms (188.×)	190
CUDA	0.10 ms	0.14 ms	0.24 ms (0.92×)	460
DiffTaichi	0.11 ms	0.15 ms	0.26 ms (1.00×)	110

Evaluation

Using gradient descent optimization on the initial velocity field, find a velocity field that changes the pattern of the fluid to a target image.

Approach	Forward Time	Backward Time	Total Time	# Essential LoC
PyTorch (CPU, f32)	405 ms	328 ms	733 ms (13.8×)	74
PyTorch (GPU, f32)	254 ms	457 ms	711 ms (13.4×)	74
Autograd (CPU, f64)	307 ms	1197 ms	1504 ms (28.4×)	51
JAX (GPU, f32)	24 ms	75 ms	99 ms (1.9×)	90
DiffTaichi (CPU, f32)	66 ms	132 ms	198 ms (3.7×)	75
DiffTaichi (GPU, f32)	24 ms	29 ms	53 ms (1.0×)	75

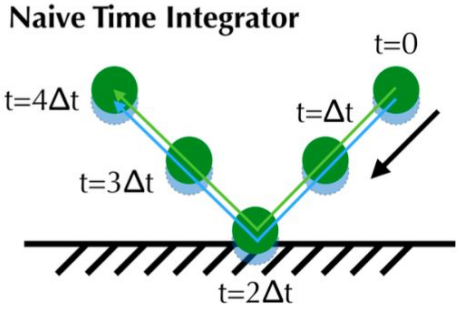
Improving Collision Detection

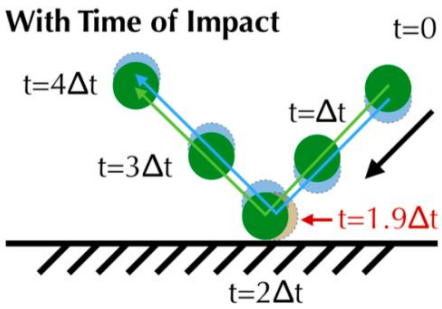
Elastic Collision
Gradient of Final Height w.r.t Initial Height of should be -1.

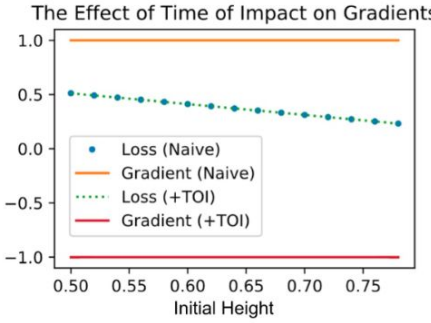
Naive Time Integrator with Discrete Collision Detection gives **Gradient = 1.**

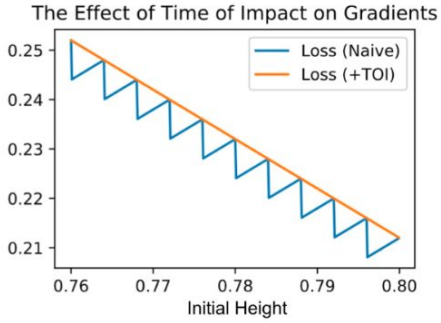
Solution
Use continuous collision detection.

Almost identical forward results (With small Δt)
Correct gradient using **Time of Impact (TOI).**

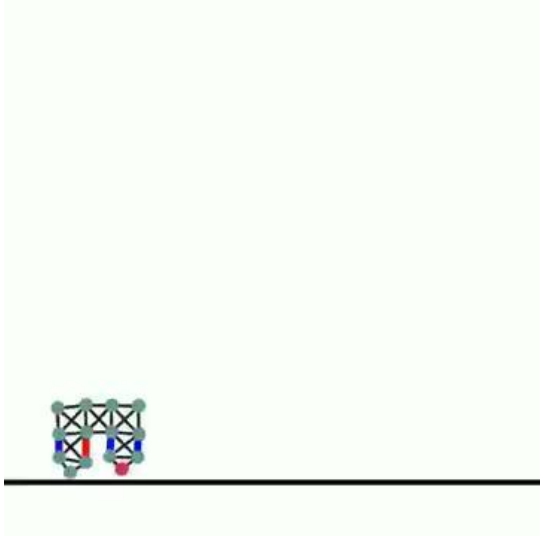


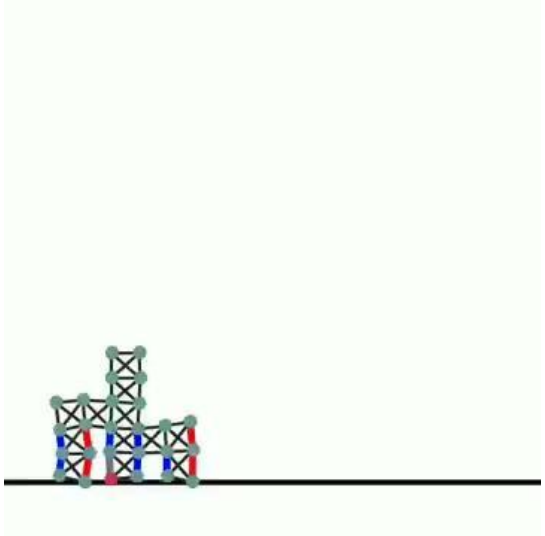


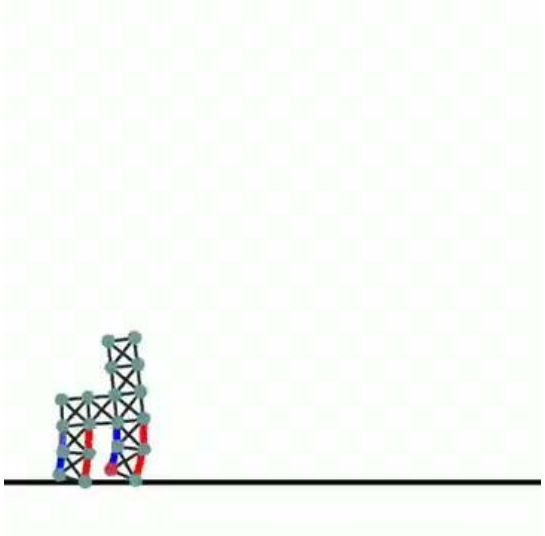




Evaluation: Rigid Body

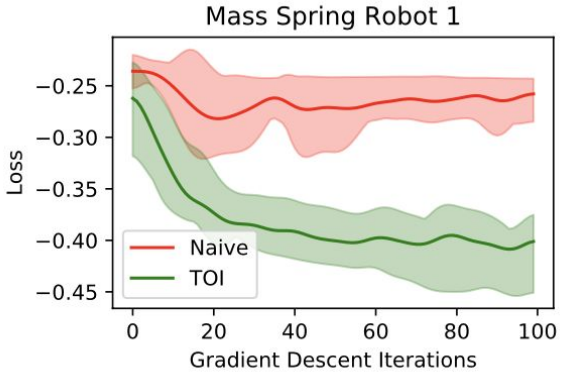


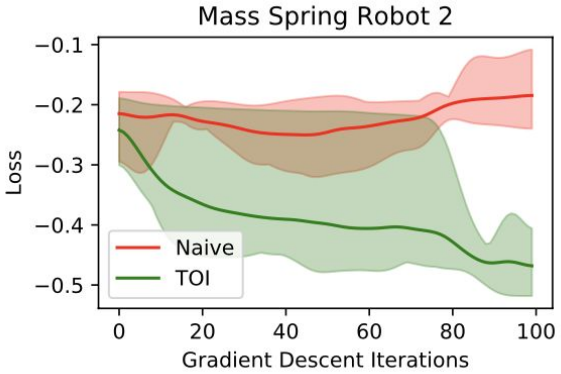


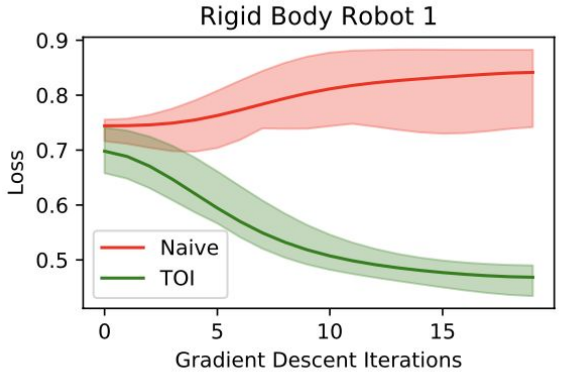


The optimization goal is to maximize the distance moved forward with 2048 time steps.

Evaluation: Rigid Body

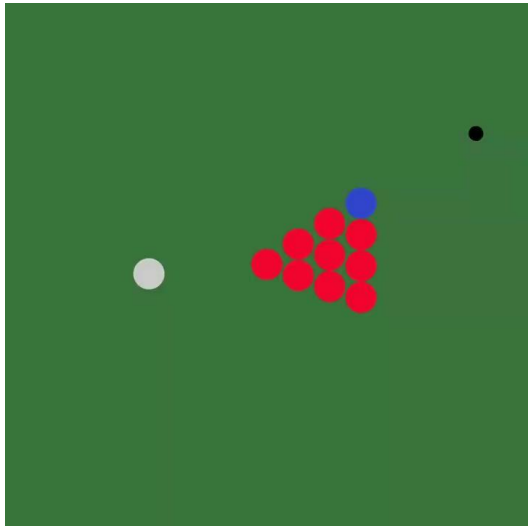




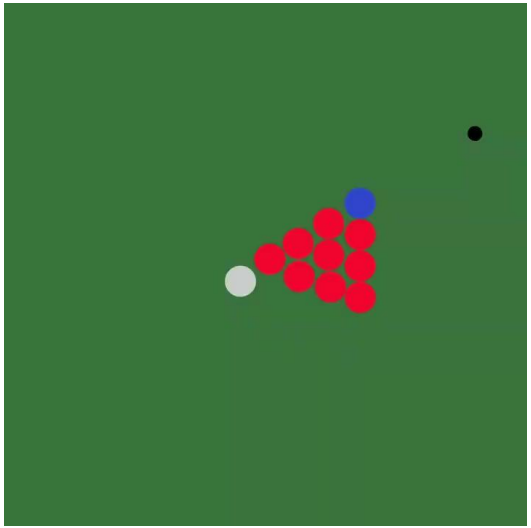


Adding TOI greatly improves gradient and optimization quality.

Results: Billiards



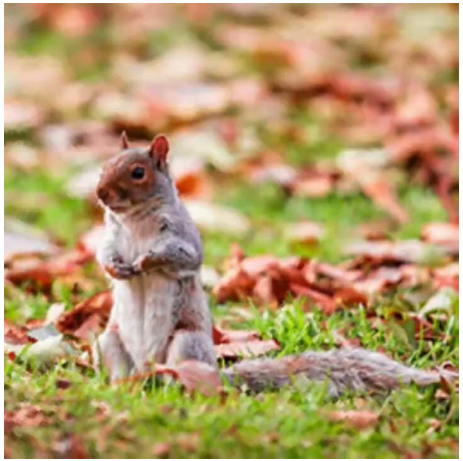
Initial

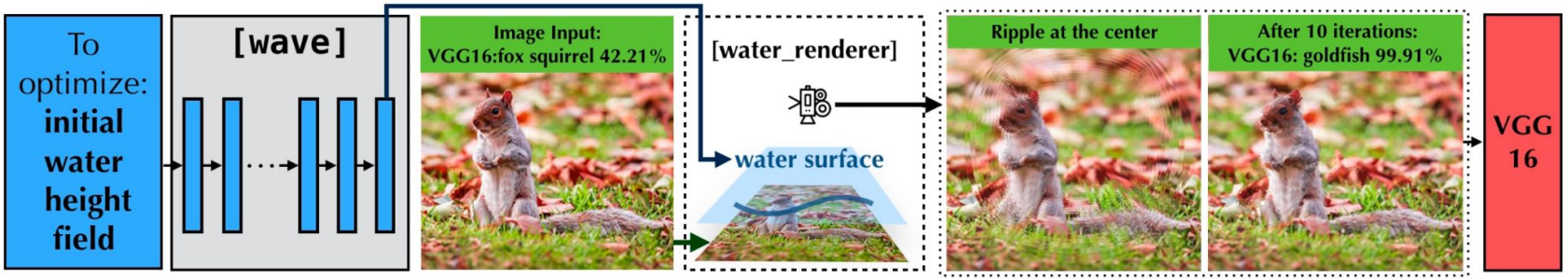


Final

Results: Water Renderer

AIM: Fool VGG-16 into thinking that the refracted squirrel image is a goldfish.





Results: Volume Renderer

Optimize for the density field of the volume to fit the target images. Using gradient descent on density field and L2 loss with target images.

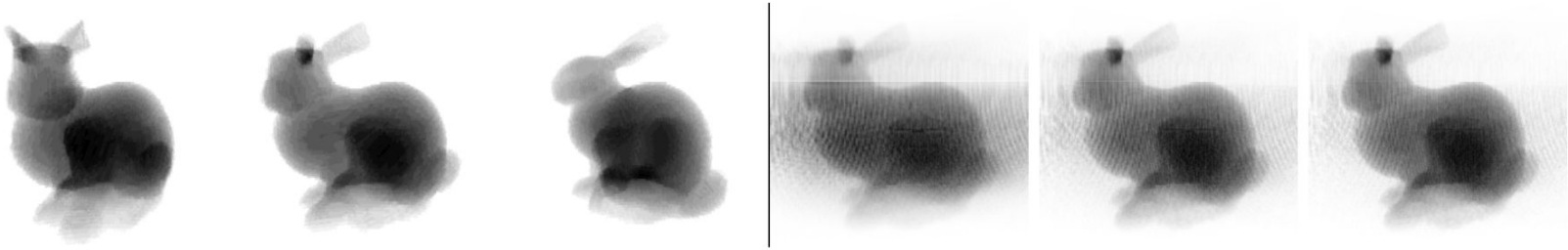


Figure 10: Volume rendering of bunny shaped density field. **Left:** 3 (of the 7) target images. **Right:** optimized images of the middle bunny after iteration 2, 50, 100. [Reproduce: **python3 volume_renderer.py**]

Additional Features

Complex Kernels
For setting the gradients of a kernel manually.

Checkpointing
Reusing allocated memory using Complex Kernels.

Forward-Mode Autodiff
Calculates Jacobian Matrix Column-Wise and returns Jacobian-Vector product. Normal Autodiff mode calculates Row-Wise Jacobian Matrix.

Limitations

1. Restrictive global data access rules. Programmer may need to make subtle changes in the original physical simulator code.
2. High memory requirements to follow global data access rules unless using complicated checkpointing.
3. Restrictive kernel structure. (Multiple Loops, or straight line statements).
4. Only the code inside the Taichi Kernels is differentiable.

Thank You!