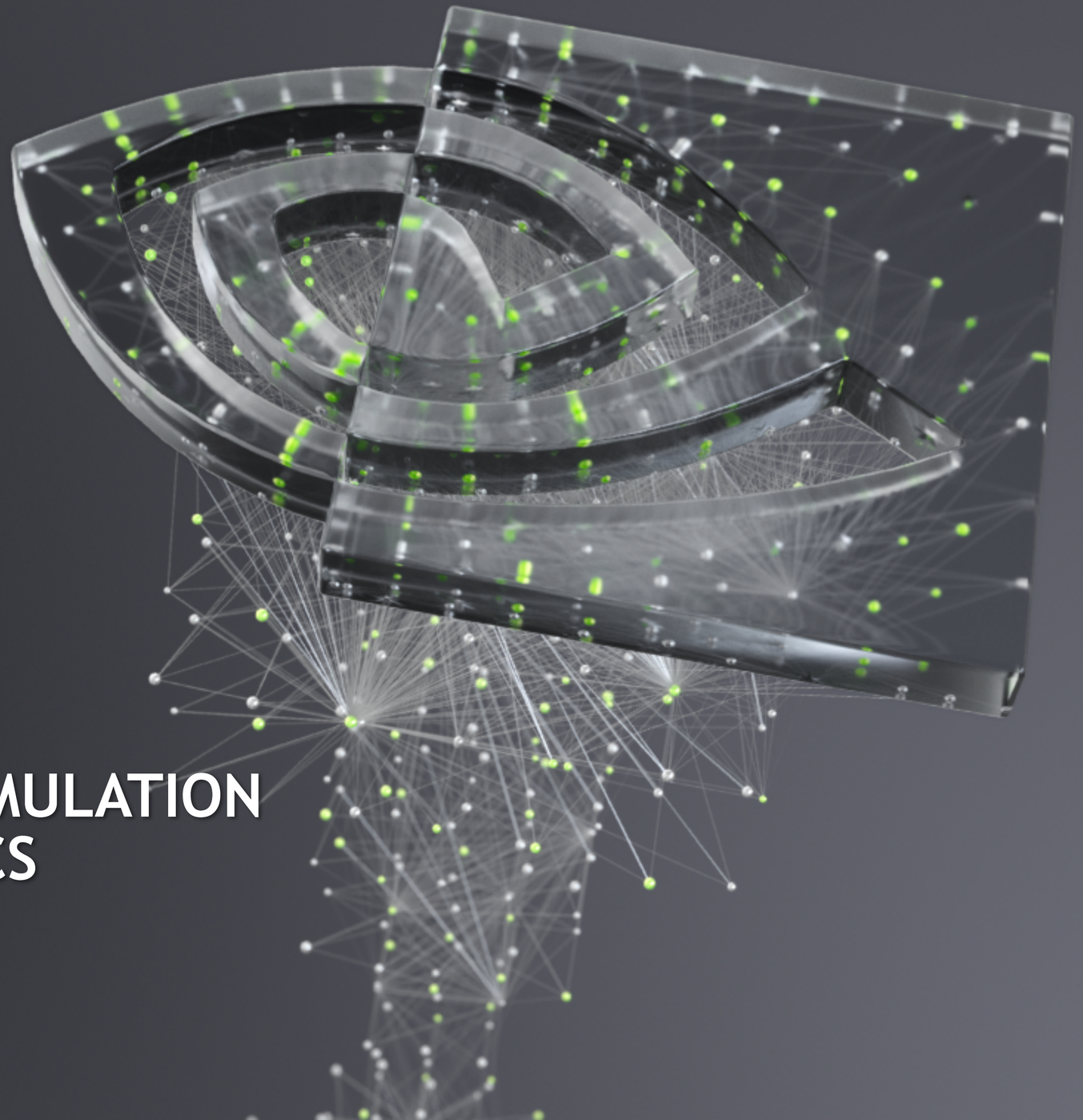


DIFFERENTIABLE PHYSICS SIMULATION FOR LEARNING AND ROBOTICS

Miles Macklin, GTC 2021

GOAL

- Minimize a scalar loss function s() w.r.t system parameters $\mathbf{x}(t_0)$

$$s\left(\mathbf{x}(t_1)\right) = s\left(\mathbf{x}(t_0) + \int_{t_0}^{t_1} f(\mathbf{x}(t))dt\right)$$

System State:

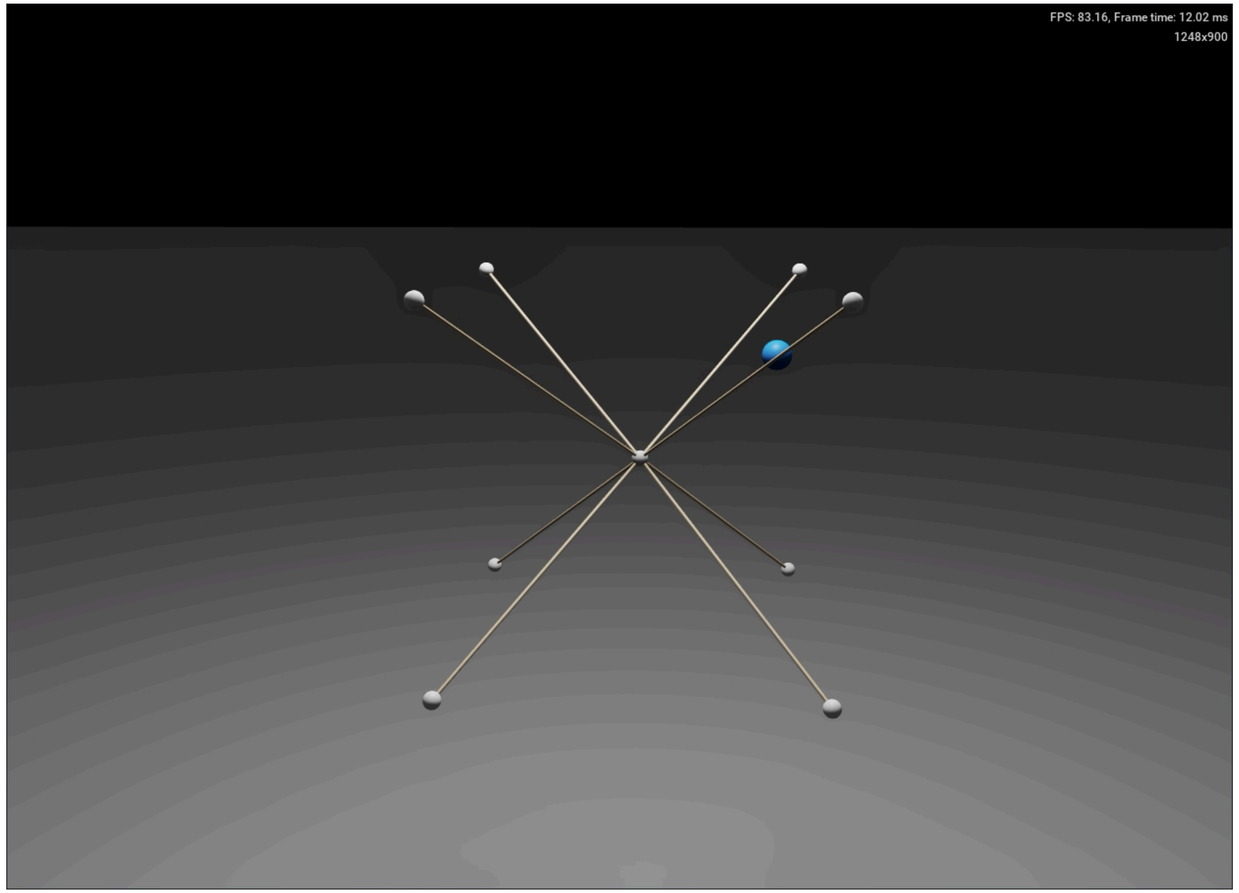
$$\mathbf{x}(t) = \begin{bmatrix} \mathbf{q} \\ \dot{\mathbf{q}} \\ \theta \end{bmatrix}$$

Forward ODE:

$$\dot{\mathbf{x}}(t) = f(\mathbf{x}(t))$$

EXAMPLE - MASS SPRING CAGE

- Minimize distance of center particle to target after 2 sec
- Optimize over spring rest lengths
- 3-4 LBFGS iterations



AUTO DIFFERENTIATION

COMPUTING GRADIENTS

- For optimization we want the gradient of scalar loss $s(\mathbf{x})$ at $t = t_0$
- Define the adjoint of a variable as $\mathbf{x}^*$
- Goal: given $\mathbf{x}^*(t_1)$ compute $\mathbf{x}^*(t_0)$

Adjoint Variable

$$\mathbf{x}^*(t) = \frac{\partial s}{\partial \mathbf{x}}^T = \begin{bmatrix} \frac{\partial s}{\partial x_1} \\ \vdots \\ \frac{\partial s}{\partial x_n} \end{bmatrix}$$

$$\mathbf{x}, \mathbf{x}^* \in \mathbb{R}^n$$

CONTINUOUS ADJOINT METHOD

- Computes gradient of scalar loss function via. reverse ODE

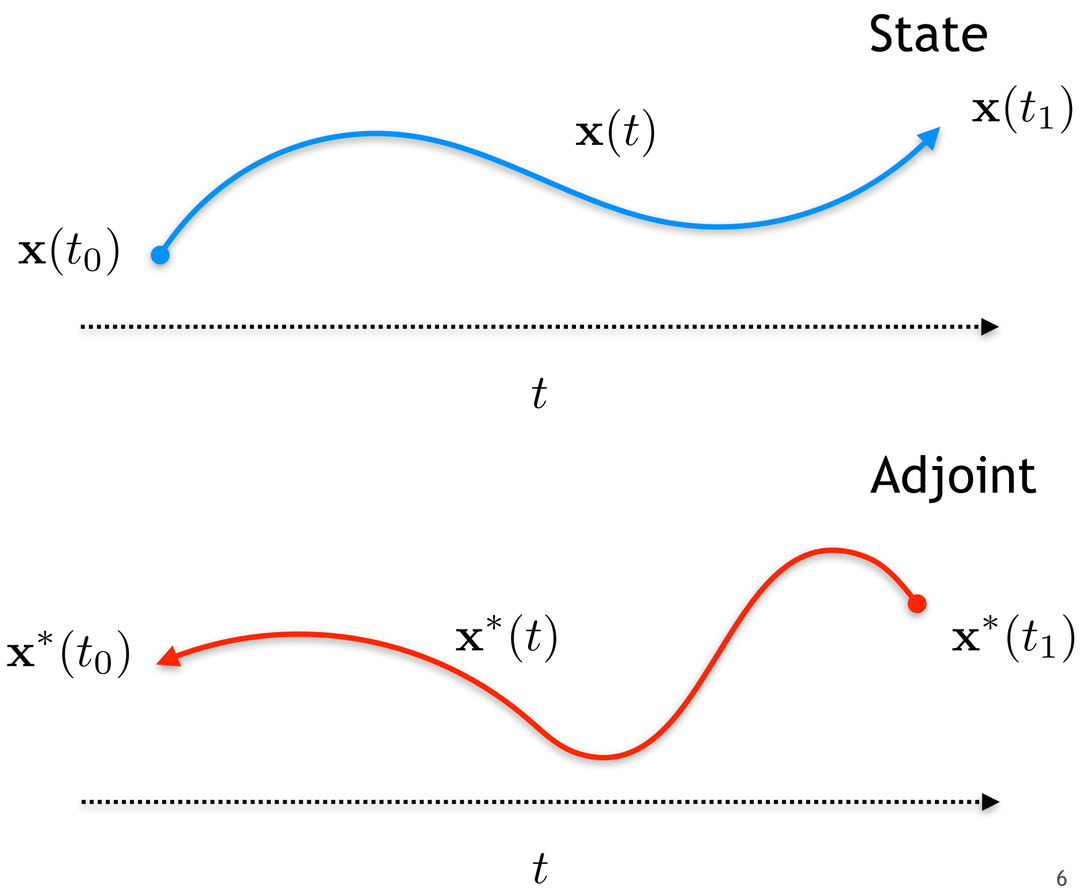
Forward ODE:

$$\dot{\mathbf{x}}(t) = f(\mathbf{x}(t))$$

Calculus of Variations

Reverse ODE:

$$\dot{\mathbf{x}}^*(t) = -\frac{\partial f}{\partial \mathbf{x}}^T \mathbf{x}^*(t)$$



DISCRETE ADJOINT METHOD

- Replace ODE with time-stepping equations:

$$\mathbf{x}^{t+1} = f(\mathbf{x}^t)$$

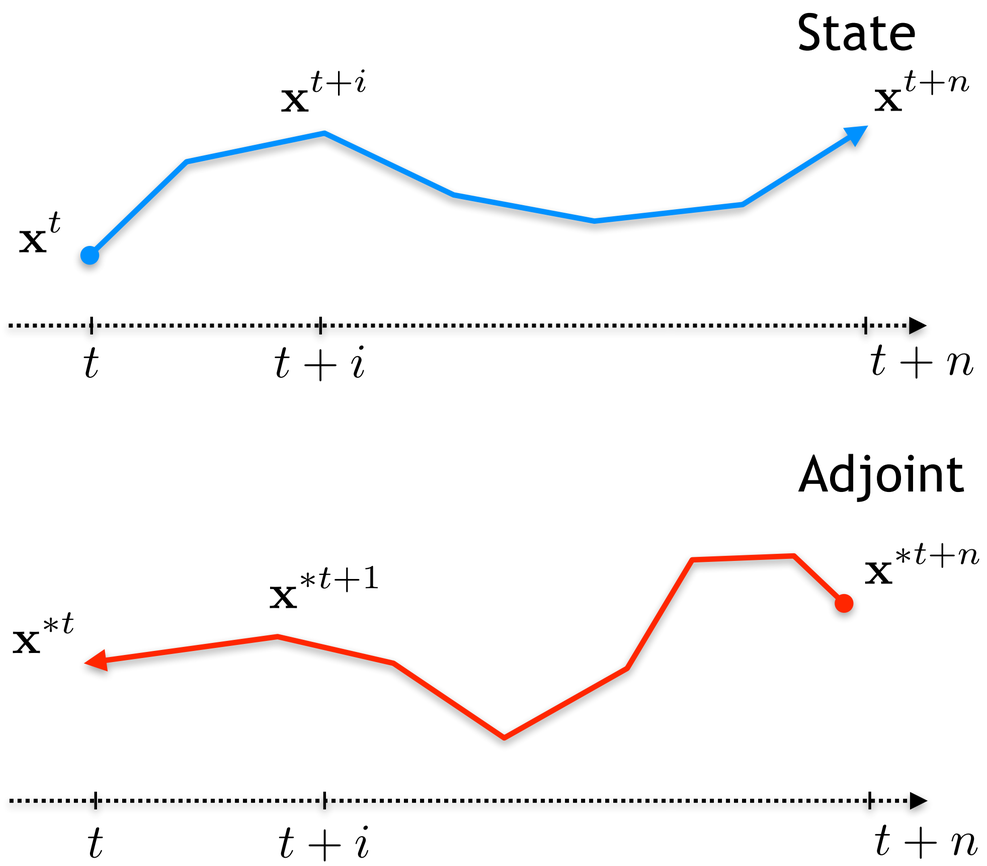
- Discrete trajectory + loss:

$$s(\mathbf{x}^{t+n}) = s(f(f(f(\mathbf{x}^t))$$

- Apply chain rule:

$$\mathbf{x}^{*^t} = \frac{\partial s}{\partial \mathbf{x}}\Big|_{t+0}^{T} = \frac{\partial f}{\partial \mathbf{x}}\Big|_{t+0}^{T} \cdot \frac{\partial f}{\partial \mathbf{x}}\Big|_{t+1}^{T} \cdot \frac{\partial f}{\partial \mathbf{x}}\Big|_{t+2}^{T} \cdot \frac{\partial s}{\partial \mathbf{x}}\Big|_{t+3}^{T}$$

- Two ways to evaluate the chain rule..



FORWARD ACCUMULATION (TANGENT MODE)

- Forward:

$$\frac{\partial s(f(g(x))}{\partial x} = \frac{\partial s}{\partial f}\left(\frac{\partial f}{\partial g}\frac{\partial g}{\partial x}\right)$$

$$s : \mathbb{R}^n \to \mathbb{R}$$
$$f : \mathbb{R}^m \to \mathbb{R}^n$$
$$g : \mathbb{R}^p \to \mathbb{R}^m$$

$$\boxed{\mathbb{R}^{1\times p}} = \boxed{\mathbb{R}^{1\times n}} \cdot \left(\boxed{\mathbb{R}^{n\times m}} \cdot \boxed{\mathbb{R}^{m\times p}} \right)$$

- Evaluate inside->out
- Simple, but large matrix multiplies are expensive
- Use forward mode when outputs >> params (e.g.: vector valued loss)

REVERSE ACCUMULATION (ADJOINT MODE)

- Reverse:

$$\frac{\partial s(f(g(x))}{\partial x} = \left(\frac{\partial s}{\partial f}\frac{\partial f}{\partial g}\right)\frac{\partial g}{\partial x}$$

$$s : \mathbb{R}^n \to \mathbb{R}$$
$$f : \mathbb{R}^m \to \mathbb{R}^n$$
$$g : \mathbb{R}^p \to \mathbb{R}^m$$

$$\boxed{\mathbb{R}^{1\times p}} = \left(\boxed{\mathbb{R}^{1\times n}} \cdot \boxed{\mathbb{R}^{n\times m}}\right) \cdot \boxed{\mathbb{R}^{m\times p}}$$

- Evaluate outside->in
- Use reverse mode when outputs << params (e.g.: scalar valued loss)

CONTINUOUS/DISCRETE SIDE-BY-SIDE

Continuous Loss:

$$s\left(\mathbf{x}(t_0) + \int_{t_0}^{t_1} f(\mathbf{x}(t))dt\right)$$

Forward ODE:

$$\dot{\mathbf{x}}(t) = f(\mathbf{x}(t))$$

Reverse ODE:

$$\dot{\mathbf{x}}^*(t) = -\frac{\partial f}{\partial \mathbf{x}}^T \mathbf{x}^*(t)$$

Discrete Loss:

$$s(\mathbf{x}^{t+n}) = s(f(f(f(\mathbf{x}^t))$$

Forward Time-stepping:

$$\mathbf{x}^{t+1} = f(\mathbf{x}^t)$$

Reverse Time-stepping:

$$\mathbf{x}^{*^{t-1}} = \frac{\partial f}{\partial \mathbf{x}}^T \mathbf{x}^{*^t}$$

ADJOINT OF A FUNCTION

- Given a function:

$$f(\mathbf{x}, \mathbf{y}) \rightarrow \mathbf{z}$$

- Define adjoint (*) as follows:

Adjoint Variable: $\mathbf{z}^* = \frac{\partial s}{\partial \mathbf{z}}^T$

$$f^*(\mathbf{x}, \mathbf{y}, \mathbf{z}^*) \rightarrow (\mathbf{x}^*, \mathbf{y}^*)$$

$$f^*(\mathbf{x}, \mathbf{y}, \mathbf{z}^*) \equiv \left(\frac{\partial f}{\partial \mathbf{x}}^T \mathbf{z}^*, \frac{\partial f}{\partial \mathbf{y}}^T \mathbf{z}^* \right)$$

- Adjoint returns derivative of scalar loss with respect to function inputs

ADJOINT EXAMPLES

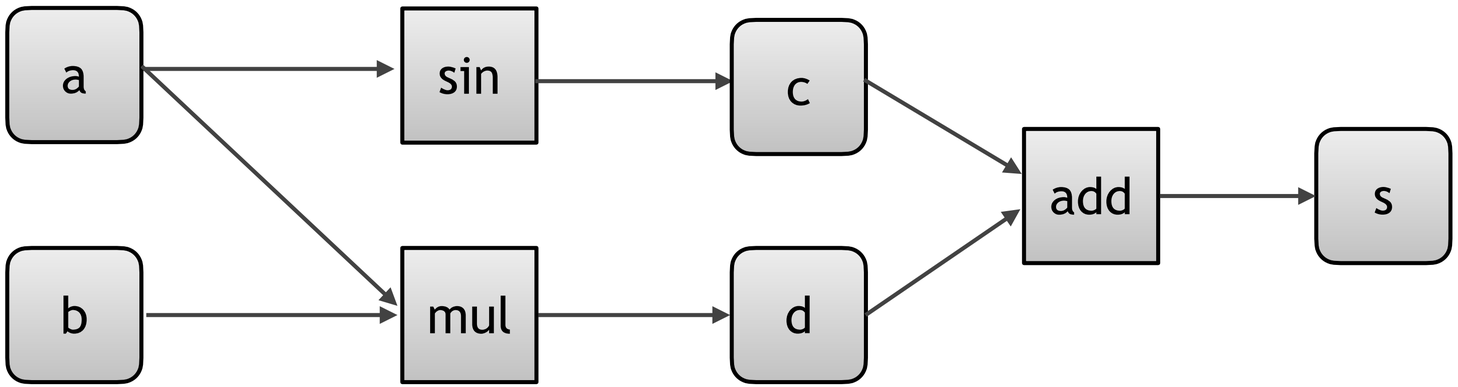
$$z = f(x, y) = x + y \qquad f^*(x, y, z^*) = [z^*, z^*]$$

$$z = f(x, y) = xy \qquad f^*(x, y, z^*) = [yz^*, xz^*]$$

$$z = f(x) = sin(x) \qquad f^*(x, z^*) = [cos(x)z^*]$$

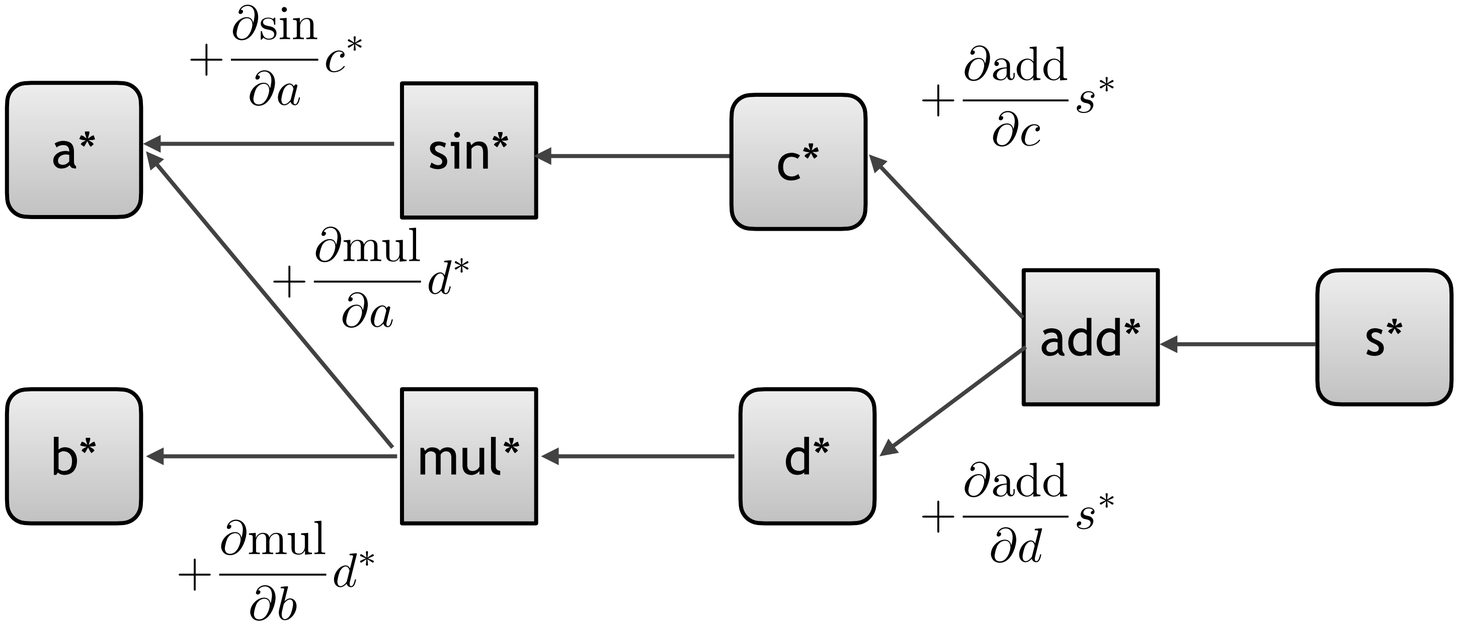
REVERSE MODE AUTO. DIFF

- Example: $s(a, b) = \sin(a) + ab$
- Forward evaluation graph:



REVERSE MODE AUTO. DIFF

- Example: $s(a,b) = \sin(a) + ab$



Solution:

$$\frac{\partial s}{\partial a} = \cos(a) + b$$
$$\frac{\partial s}{\partial b} = a$$

Adjoint Variables:

$$a^* = \frac{\partial s}{\partial a}^T$$

Seed Variable:

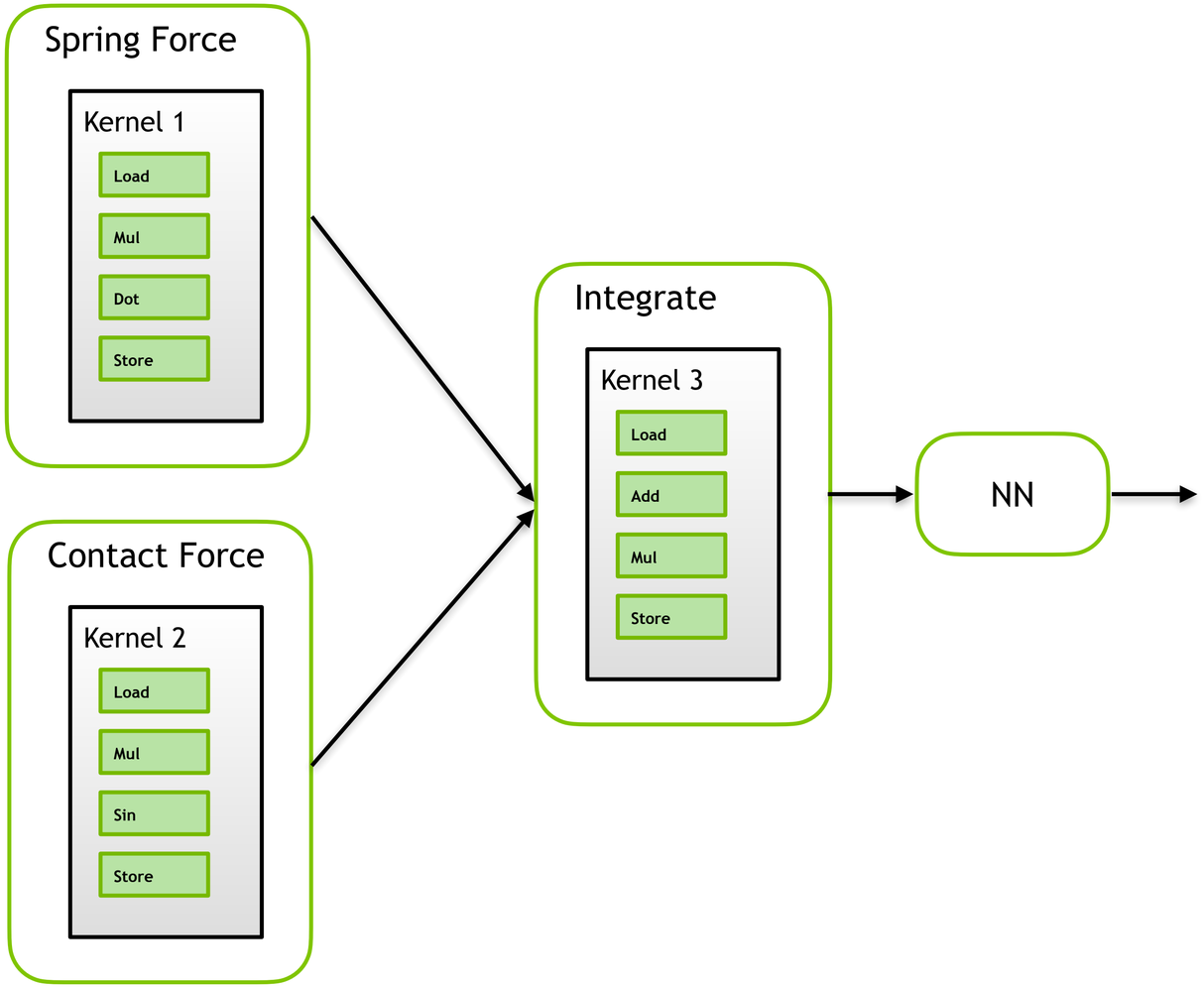
$$s^* = \frac{\partial s}{\partial s}^T = 1$$

AUTODIFF FRAMEWORKS

- Graph Evaluation
  - Runtime
  - Functional, tensor centric
  - PyTorch, TensorFlow
- Program Transformation
  - Compile time
  - Imperative, thread centric
  - DiffTaichi, Google Tangent, Tapenade, dFlex
- Symbolic
  - Expression rewriting
  - Matlab, Mathematica, Maple

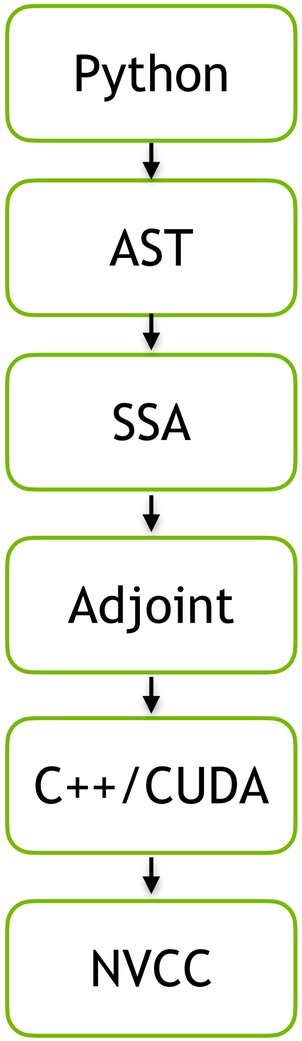
3-LEVEL AUTO DIFF

- Top level
  - computation graph + tape
  - e.g.: loss functions, NN model
- Middle level
  - forward/backward kernels
  - e.g.: force evaluation
- Bottom level
  - mathematical primitives
  - e.g.: sin, cos, dot, cross, etc



PROGRAM TRANSFORMATION

- Middle level auto-diff
- Given abstract syntax tree generate a function's adjoint:
  - Traverse tree (`import ast`)
  - Convert to static single assignment (SSA) form
  - Run function forward (recording state)
  - Run function backward (accumulate gradients)



SIMPLE EXAMPLE

- Python->C++ SSA
- State is local in registers
- Kernel fusion is implicit
- Flexible indexing
- Gather/Scatter Ops
- Runtime JIT compilation

```python
@df.func
def simple(a : float, b: float):
    return df.sin(a) + a*b
```

↓

```cpp
void simple_reverse(
    float var_a,
    float var_b,
    float& adj_a,
    float& adj_b,
    float adj_s)
{
    //---------
    // dual vars
    float adj_0 = 0.0f;
    float adj_1 = 0.0f;
    float adj_2 = 0.0f;
    //---------
    // forward
    float var_0 = df::sin(var_a);
    float var_1 = df::mul(var_a, var_b);
    float var_2 = df::add(var_0, var_1);
    //---------
    // reverse
    df::adj_add(var_0, var_1, adj_0, adj_1, adj_2);
    df::adj_mul(var_a, var_b, adj_a, adj_b, adj_1);
    df::adj_sin(var_a, adj_a, adj_s);
}
```

COMPLEX KERNEL

- Triangle bending force kernel
- Many nested expressions
- Random access loads/stores
- Atomic-add accumulation

```
@df.kernel
def eval_bending(x : df.tensor(df.float3),
                 v : df.tensor(df.float3),
                 indices : df.tensor(int),
                 rest : df.tensor(float),
                 ke : float,
                 kd : float,
                 f : df.tensor(df.float3)):

    tid = df.tid()

    # triangle indices
    i = df.load(indices, tid*4+0)
    j = df.load(indices, tid*4+1)
    k = df.load(indices, tid*4+2)
    l = df.load(indices, tid*4+3)

    rest_angle = df.load(rest, tid)

    # load positions
    x1 = df.load(x, i)
    x2 = df.load(x, j)
    x3 = df.load(x, k)
    x4 = df.load(x, l)

    # load velocities
    v1 = df.load(v, i)
    v2 = df.load(v, j)
    v3 = df.load(v, k)
    v4 = df.load(v, l)

    n1 = df.cross(x3-x1, x4-x1)
    n2 = df.cross(x4-x2, x3-x2)

    n1_length = df.length(n1)
    n2_length = df.length(n2)

    rcp_n1 = 1.0/n1_length
    rcp_n2 = 1.0/n2_length
    cos_theta = df.dot(n1, n2)*rcp_n1*rcp_n2

    n1 = n1*rcp_n1*rcp_n1
    n2 = n2*rcp_n2*rcp_n2

    e = x4-x3
    e_hat = df.normalize(e)
    e_length = df.length(e)

    s = df.sign(df.dot(df.cross(n2, n1), e_hat))
    angle = df.acos(cos_theta)*s

    d1 = n1*e_length
    d2 = n2*e_length
    d3 = n1*df.dot(x1-x4, e_hat) + n2*df.dot(x2-x4, e_hat)
    d4 = n1*df.dot(x3-x1, e_hat) + n2*df.dot(x3-x2, e_hat)

    # elastic forces
    f_elastic = ke*(angle - rest_angle)

    # damping forces
    f_damp = kd*(df.dot(d1, v1) + df.dot(d2, v2) + df.dot(d3, v3) + df.dot(d4, v4))

    # total force, proportional to edge length
    f_total = 0.0 - e_length*(f_elastic + f_damp)
    df.atomic_add(f, i, d1*f_total)
    df.atomic_add(f, j, d2*f_total)
    df.atomic_add(f, k, d3*f_total)
    df.atomic_add(f, l, d4*f_total)
```

PROGRAM TRANSFORMATION - ALGORITHM

- Given an AST
- Recursive program to generate the adjoint of a function
- Assume that each node knows how to compute f, f*

```
forward_ops = []
reverse_ops = []

eval(AST.Node):

    # evaluate inputs
    inputs = []
    for each input i in node.f:
        inputs.push_back(eval(i))

    # output
    forward_ops.push_back{var_n = node.f(inputs)}
    reverse_ops.push_front{[adj_j, adj_k, …] += node.fadj(inputs)}

    return var_n
```

VERIFICATION

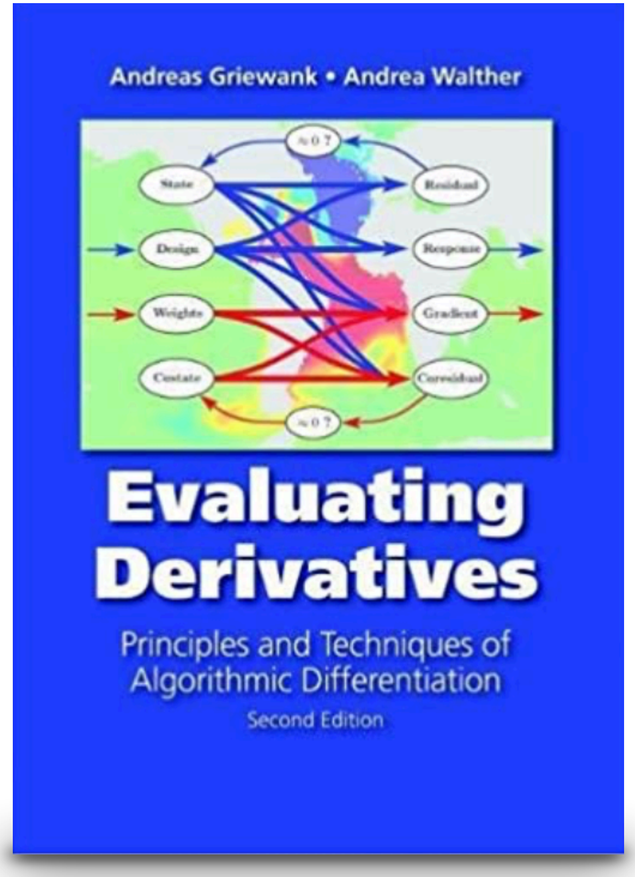
- Check gradients via. finite differencing
- torch.autograd.gradcheck
- Call function adjoint with each basis vector to evaluate full Jacobian

$$\mathbf{J}_i^T = f^*(\delta_i)$$

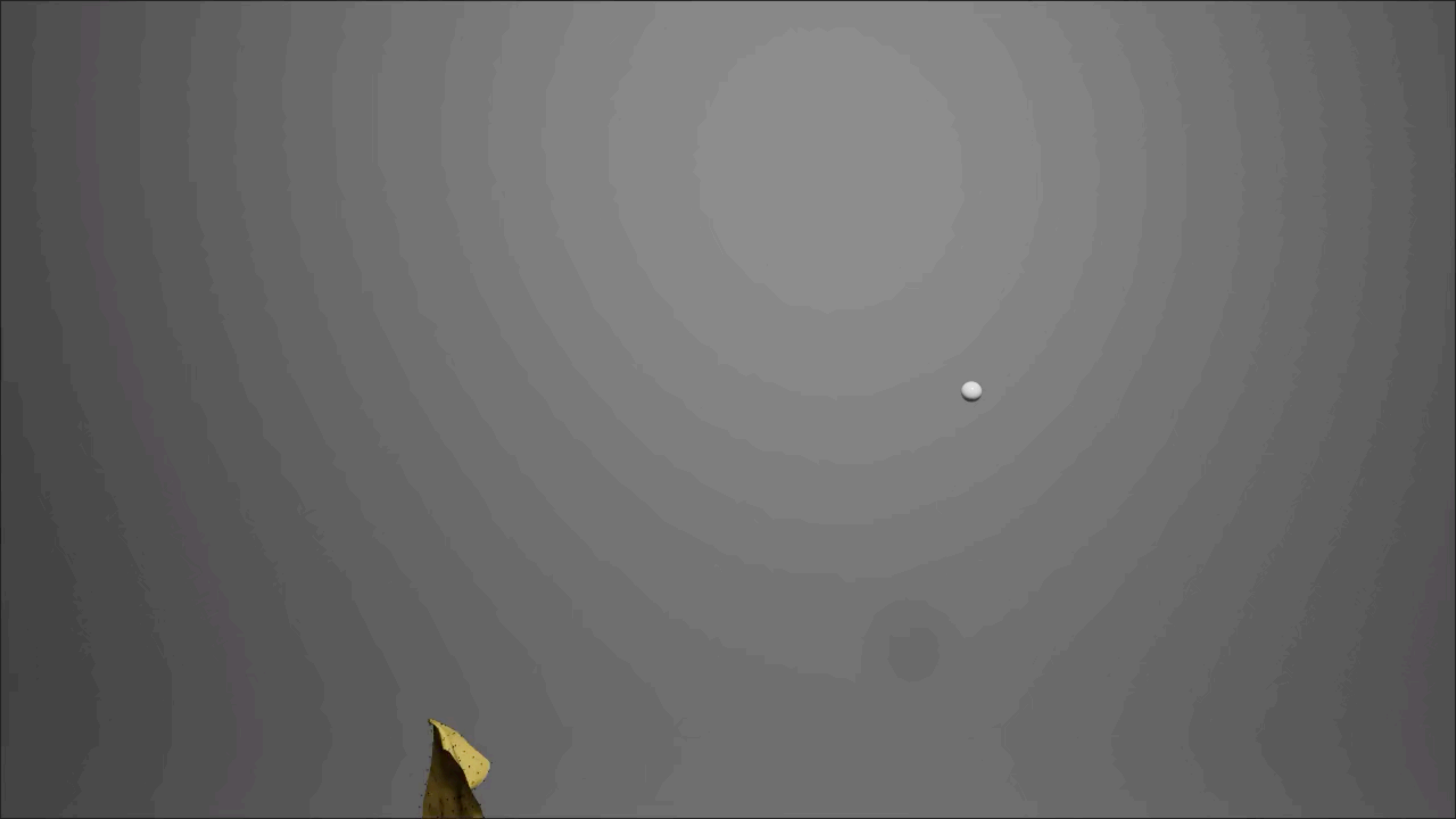
- *n* calls, one for each output

FURTHER READING

- Taichi [Hu et al. 2019]
- Evaluating Derivatives [Griewank & Walther 2008]
- Covers program transformation approach in detail
- Many more optimizations possible
- I rely on NVCC to do the heavy lifting

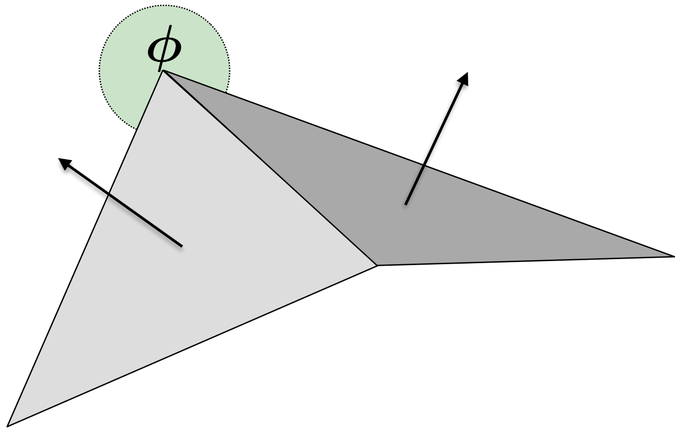


SIMULATION



EXAMPLE - THIN SHELLS

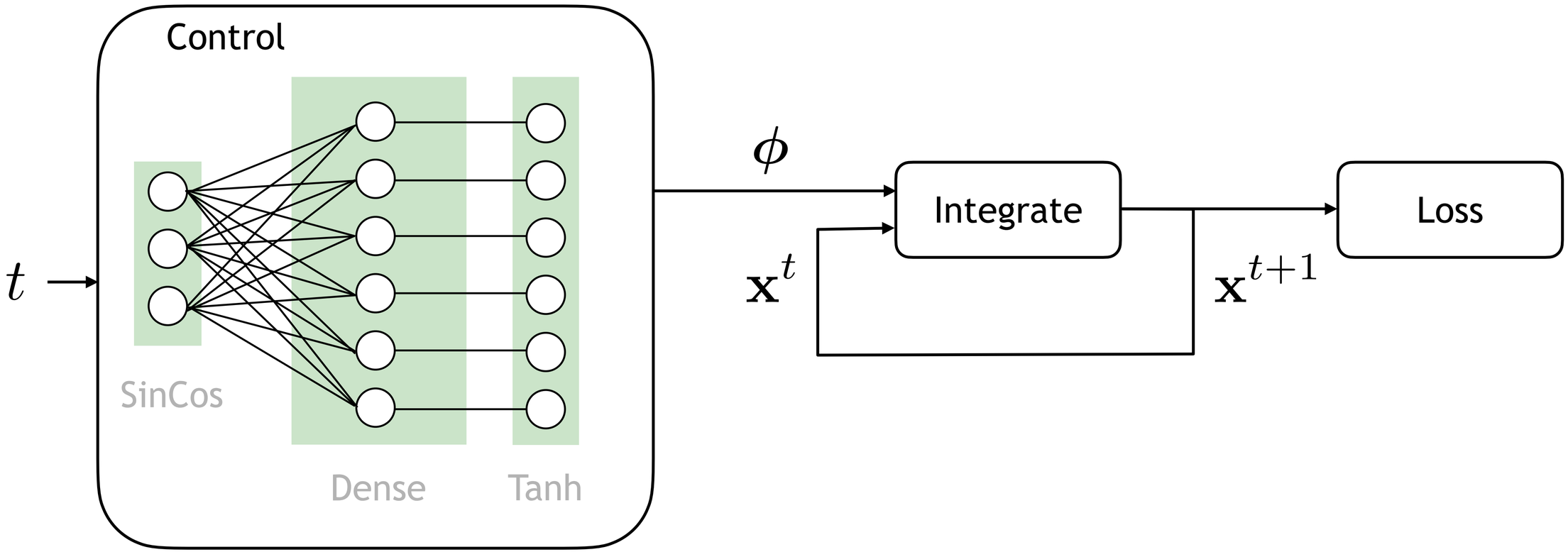
- Thin-shell FEM
- 2D NeoHookean + bending energy
- Activations into bending energy
- Lift + Drag model



[Smith et al. 2018]

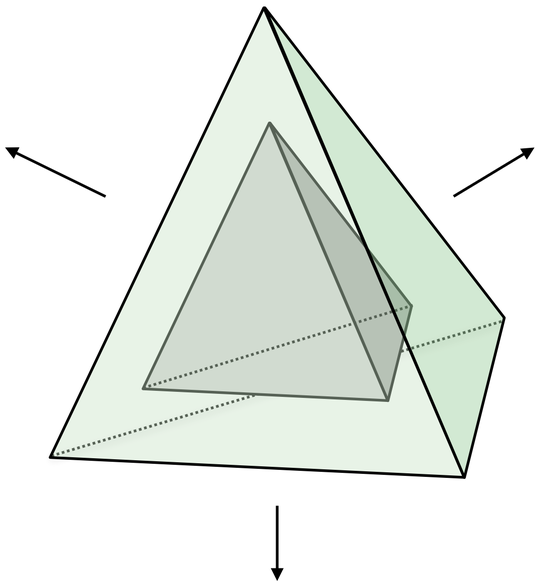
[Bridson et al. 2002]

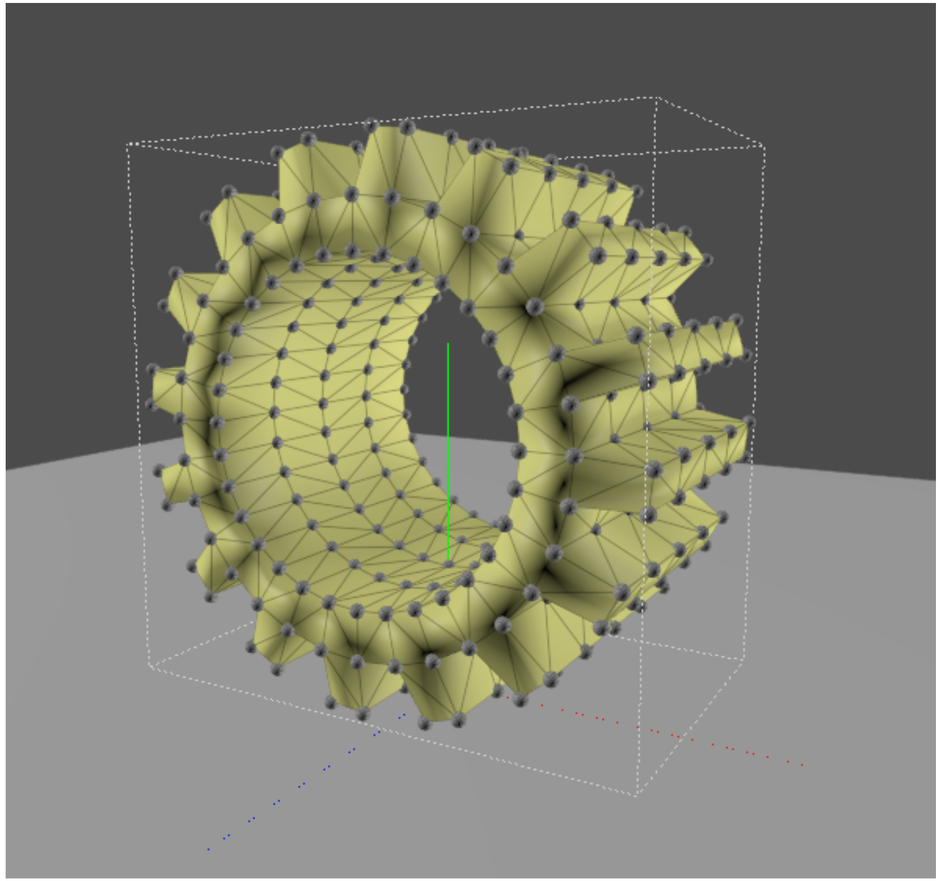
EXAMPLE - OPEN LOOP CONTROL



EXAMPLE - SOLID FEM

- Pixar's stable NeoHookean model
- Tetrahedral elements
- Volumetric activations





File Edit Create Window Testing Utilities Help

Viewport

Camera RTX Real-time

Root Layer

FPS: 66.12, Frame time: 15.13 ms
926x545

Y
Z X

Content Console Test Runner Movie Capture USD Paths

Search

This PC +

Common... Post Pro... Real-Tim... Path Tra... Layers Stage

Search

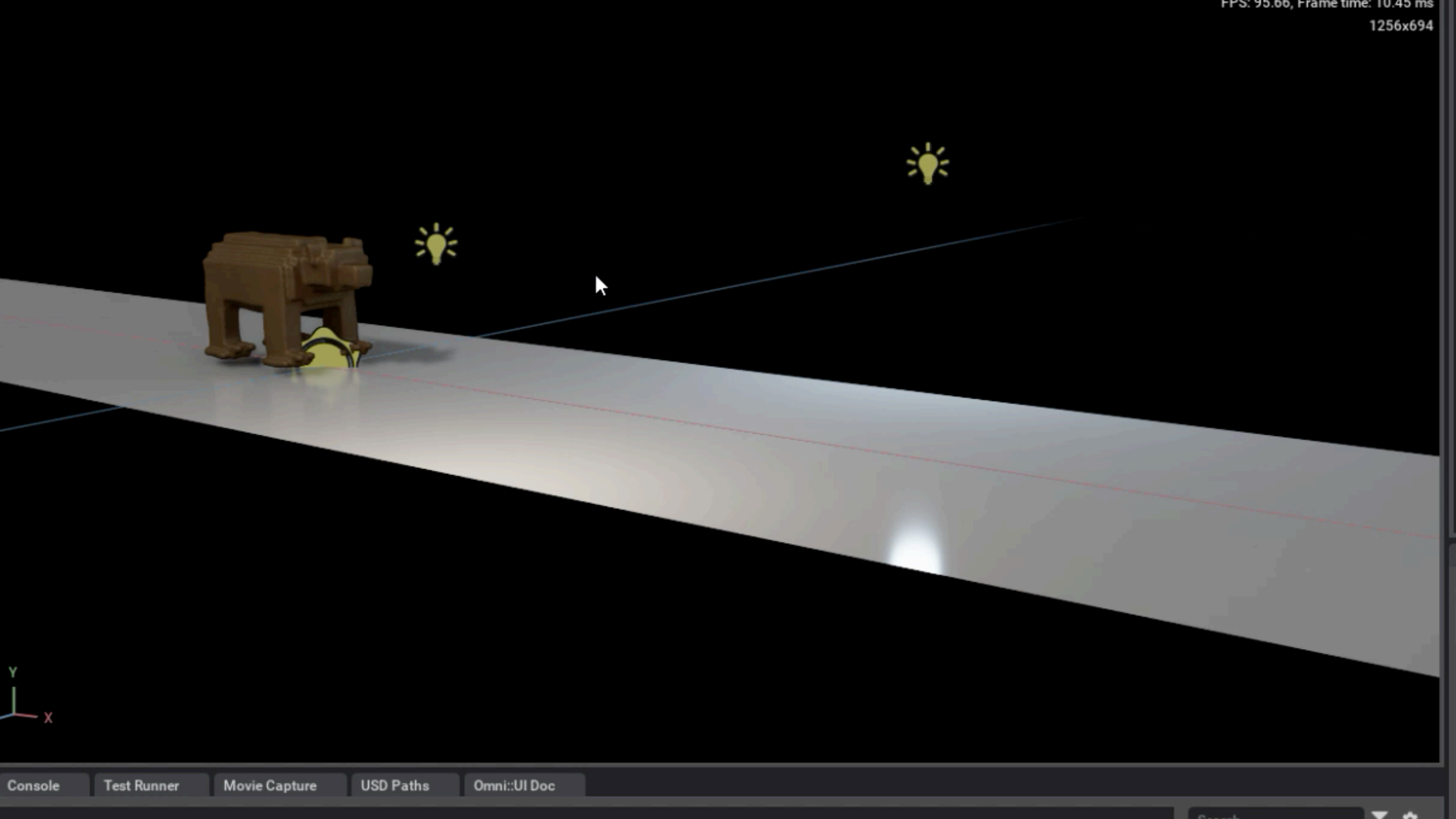
Name	Type
mesh_3	Mesh
Xform	Xform
scene	Xform
particle_instancer	PointInstancer
cloth	Mesh
plane_0	Mesh
Looks	Scope
OmniGlass	Material
OmniPBR	Material
OmniPBR_Opacity	Material
OmniPBR_01	Material
RectLight	RectLight
RectLight_01	RectLight
Camera	Camera

Details

Root Layer (selected)

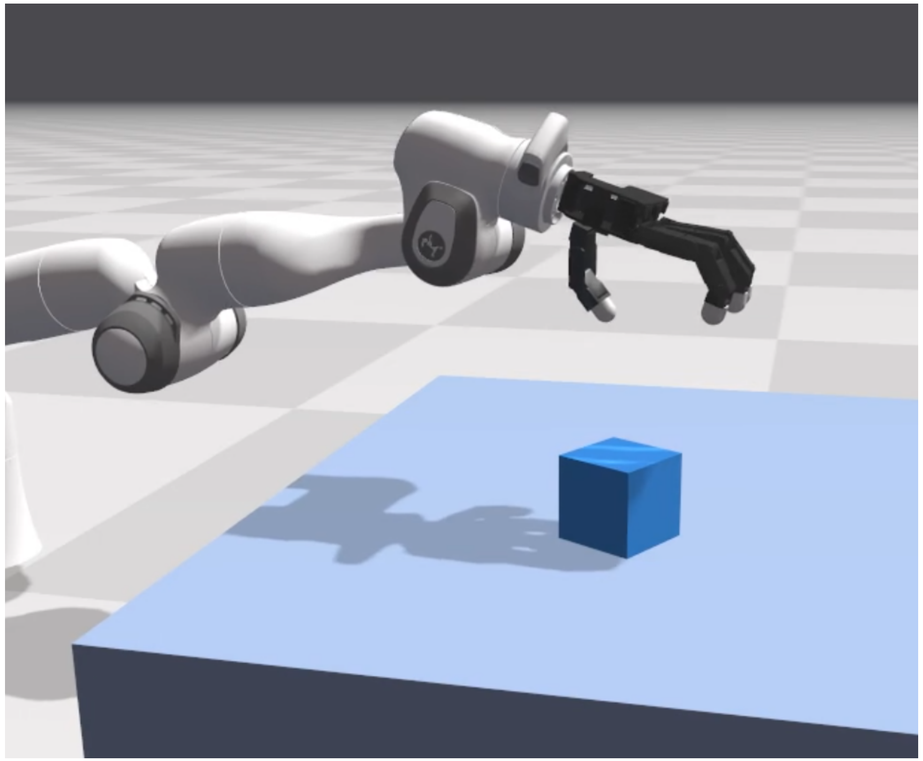
Properties

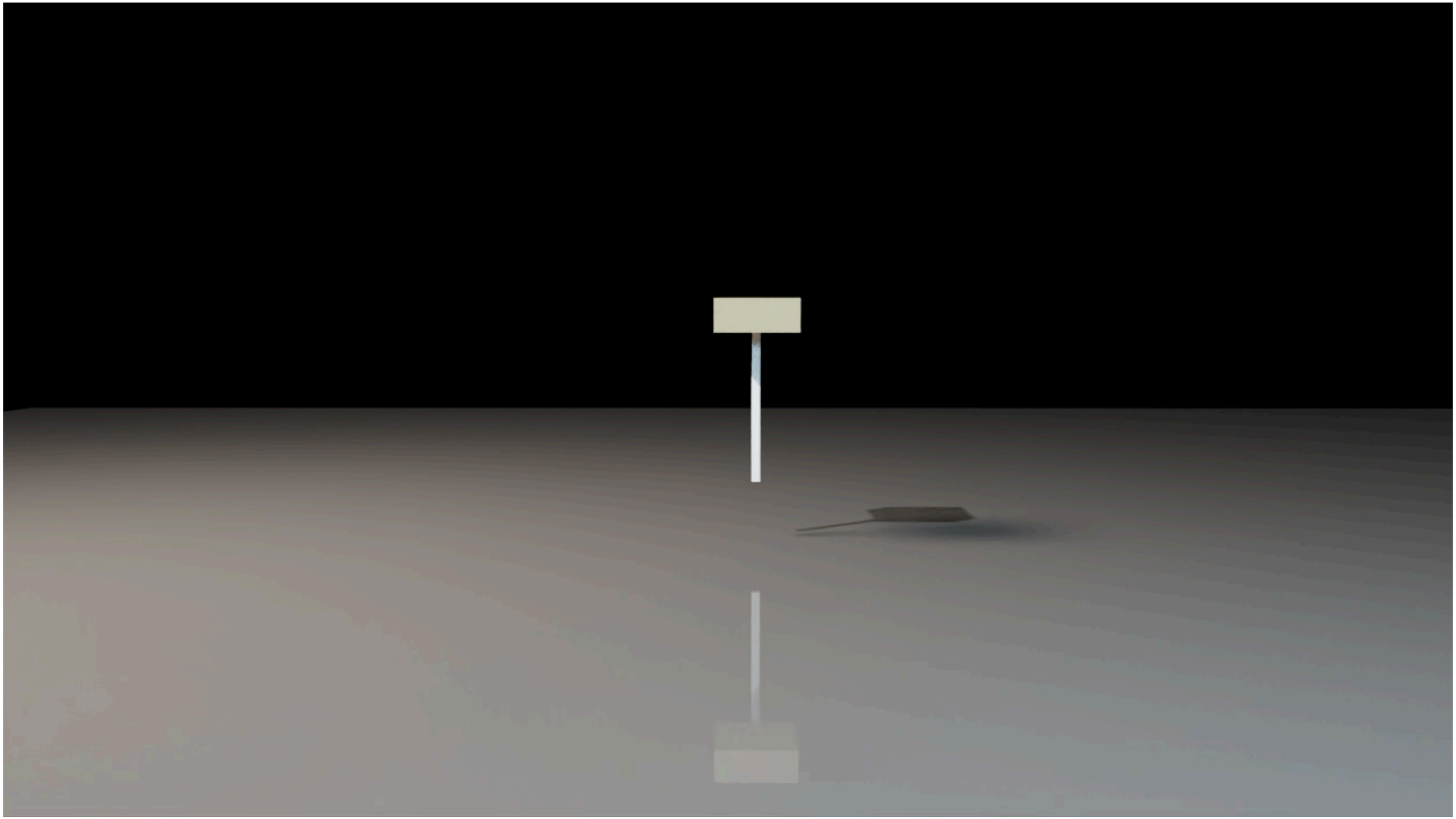
f:/gitlab/dflex/tests/outputs/top_track.usda


FPS: 95.66, Frame time: 10.45 ms
1256x694
Y
X
Console
Test Runner
Movie Capture
USD Paths
Omni::UI Doc

RIGID ARTICULATIONS

- Reduced coordinate Featherstone (CRBA)
- Prismatic, Revolute, Spherical, Fixed, Free
- Joint / MTU-based actuation
- URDF, MJCF import





File Edit Create Window Physics Rendering OmniGraph Testing IK Animation Utilities Help

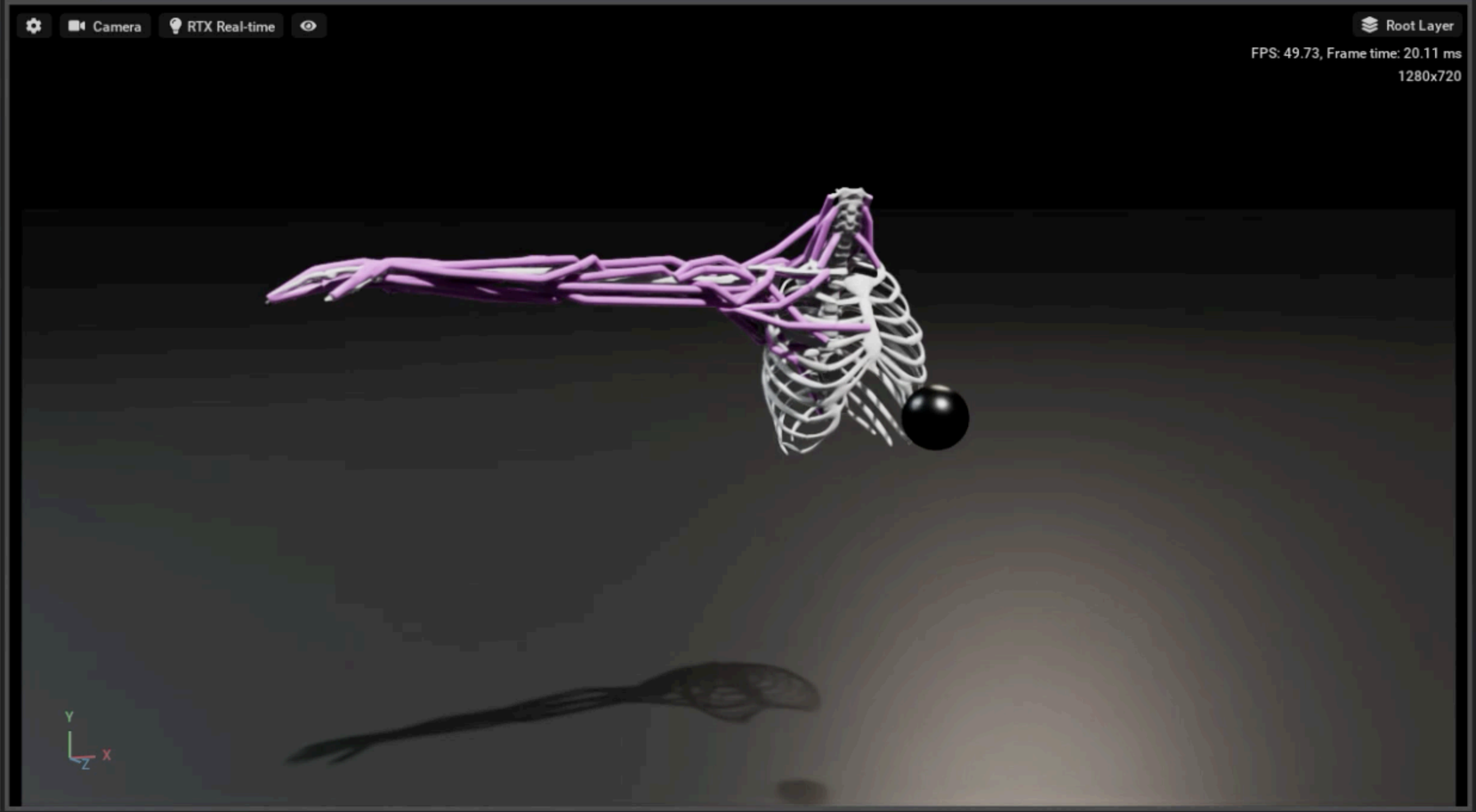
CACHE: OFF LIVE SYNC: OFF

Viewport

Camera RTX Real-time

Root Layer

FPS: 49.73, Frame time: 20.11 ms
1280x720



Content Console USD Paths Test Runner Movie Capture

Search

- Omniverse
  - localhost (signed out)
  - Add New Connection
- This PC
  - [Desktop]
  - [Documents]
  - [Downloads]
  - [Pictures]
  - C:
  - E:
  - F:
  - H:

Layer Stage Real-Time Mode Settin... Path-Traced Mode Sett... Common Rendering Set...

Indirect Diffuse Light

Ambient Light Color	R:0.100 G:0.100 B:0.100	Reset
Ambient Light Intensity	0.500	Reset
Enable Ambient Occlusion (AO)	✓	Reset
AO - Square root of temporal window length	3	Reset
AO - Minimum Samples per Pixel	1	Reset
AO - Maximum Samples per Pixel	9	Reset
AO - Ray Length	250.000	Reset
Enable Indirect Diffuse GI	✓	Reset
Enable Cached GI		Reset
Cached GI - Energy Preserving	✓	Reset
Cached GI - Samples Per Pixel	1	Reset
Indirect Diffuse GI - Intensity	1.000	Reset
Indirect Diffuse GI - Max Bounces	2	Reset
Indirect Diffuse GI - PSTF Cache Update Tile	10	Reset
Indirect Diffuse GI - Lower Resolution		Reset

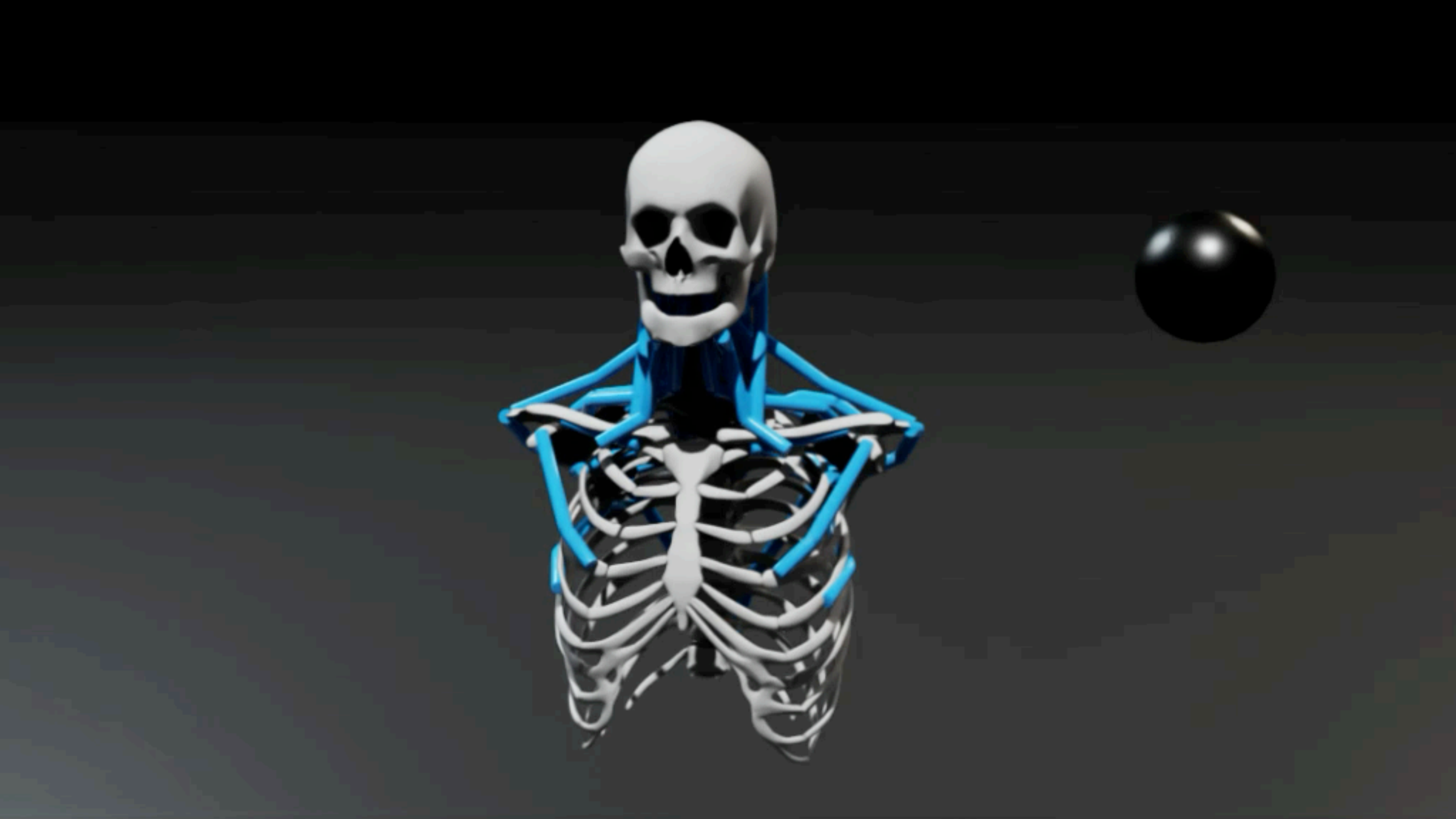
Multi-GPU

Enable Multi-GPU		Reset
Tile Overlap	32	Reset

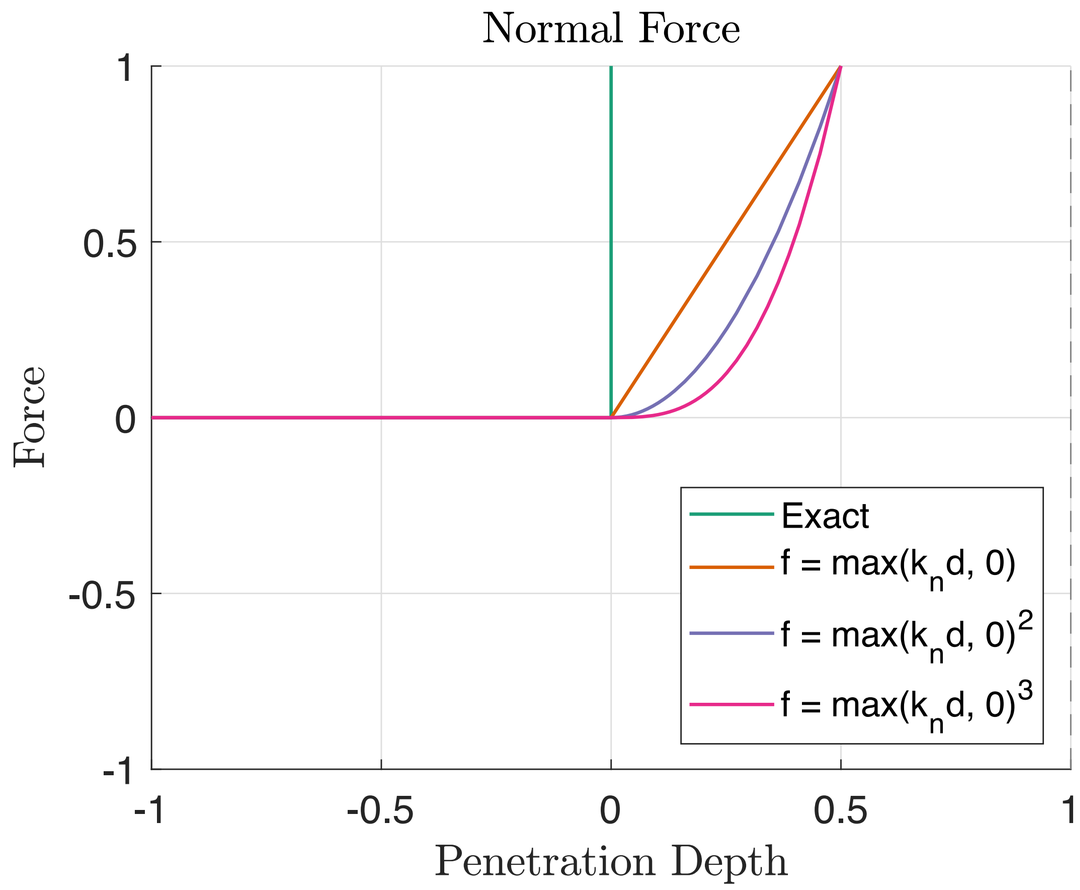
Details PhysX Properties Audio Settings

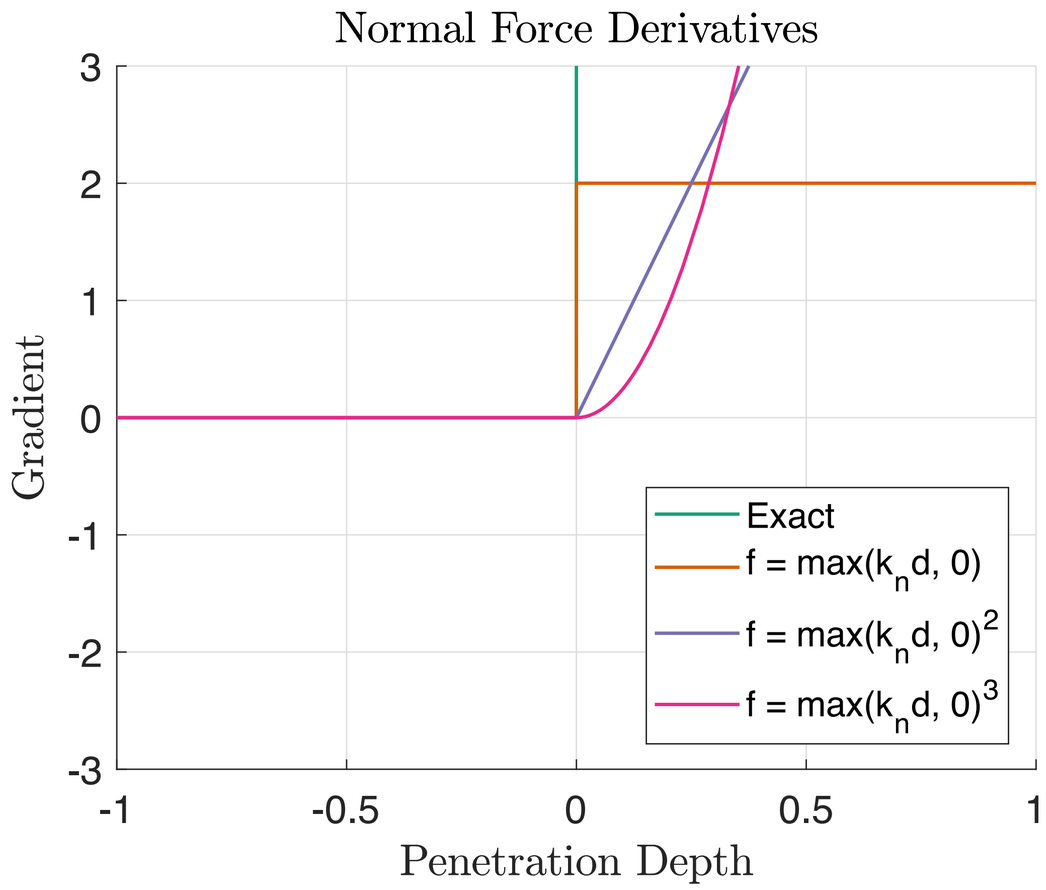
(nothing selected)

Select USD Primitive or Layer

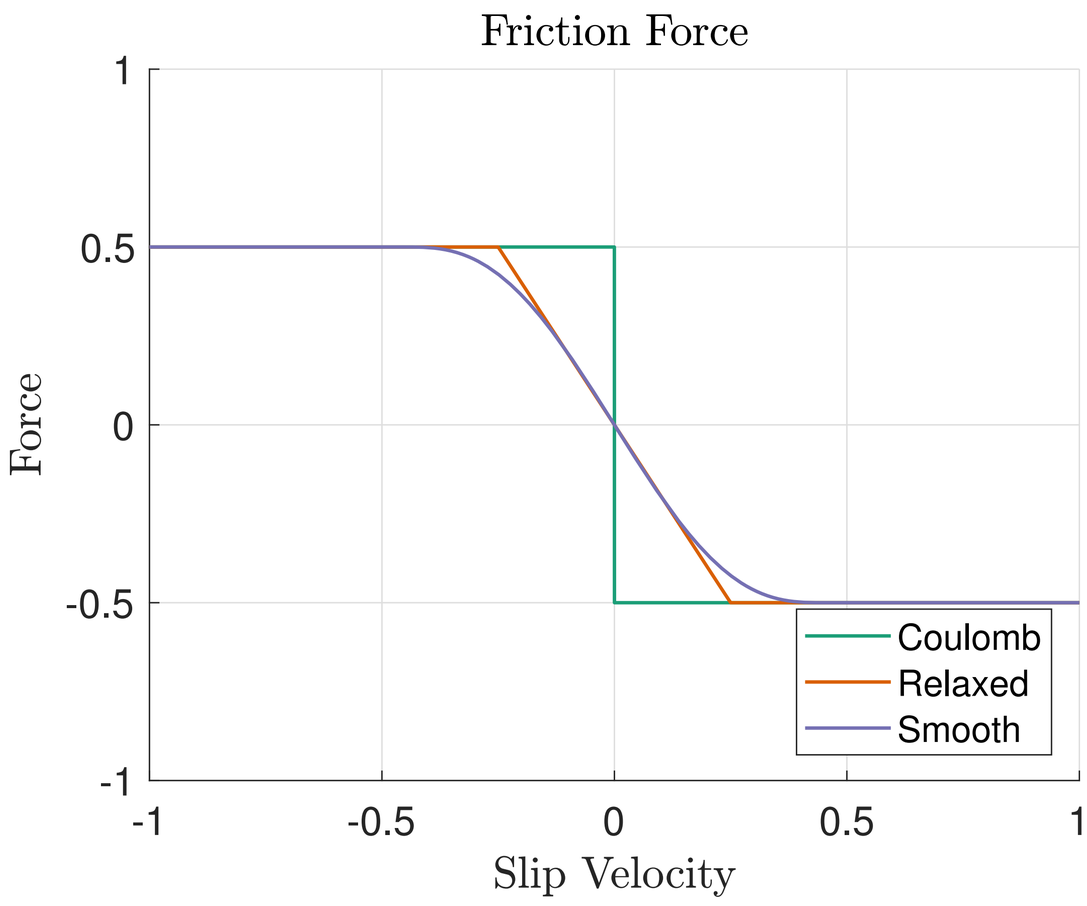


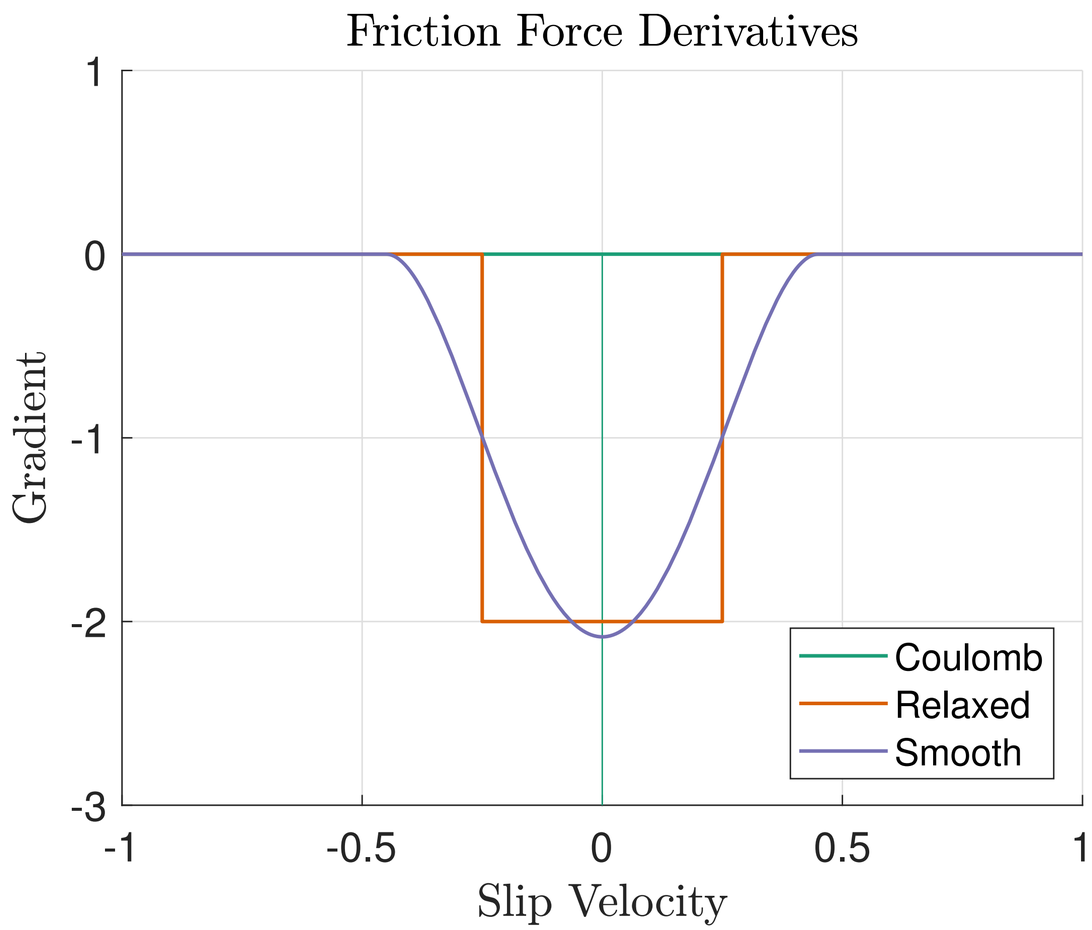
CONTACT SMOOTHNESS

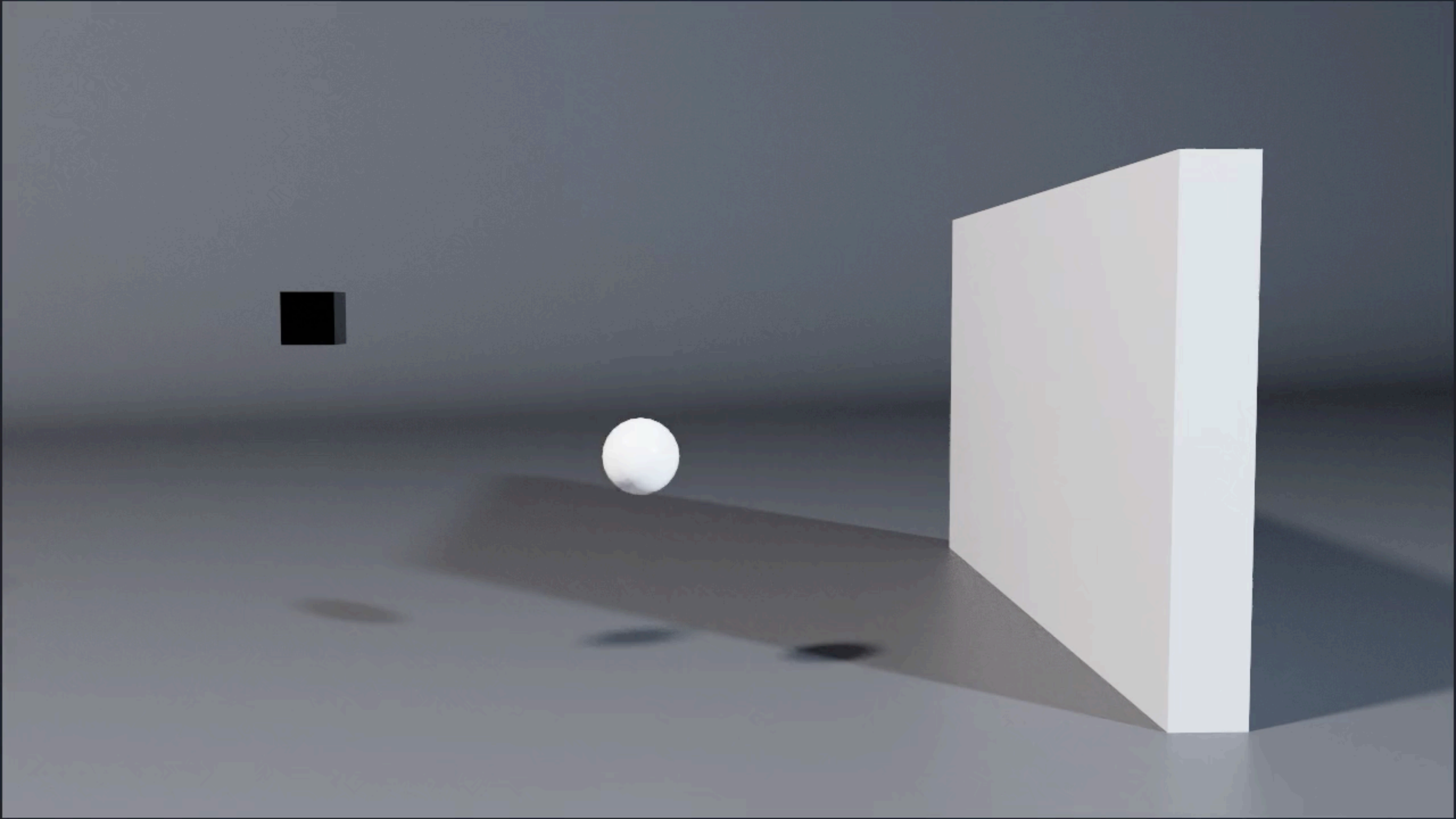




FRICTION SMOOTHNESS

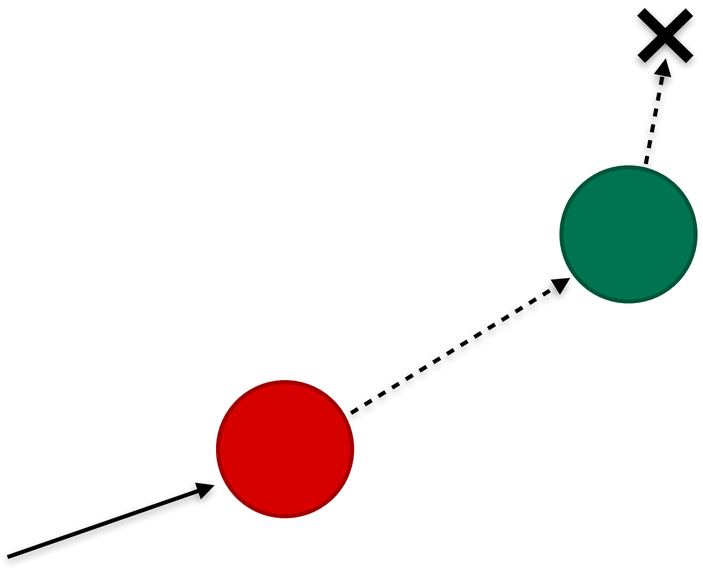


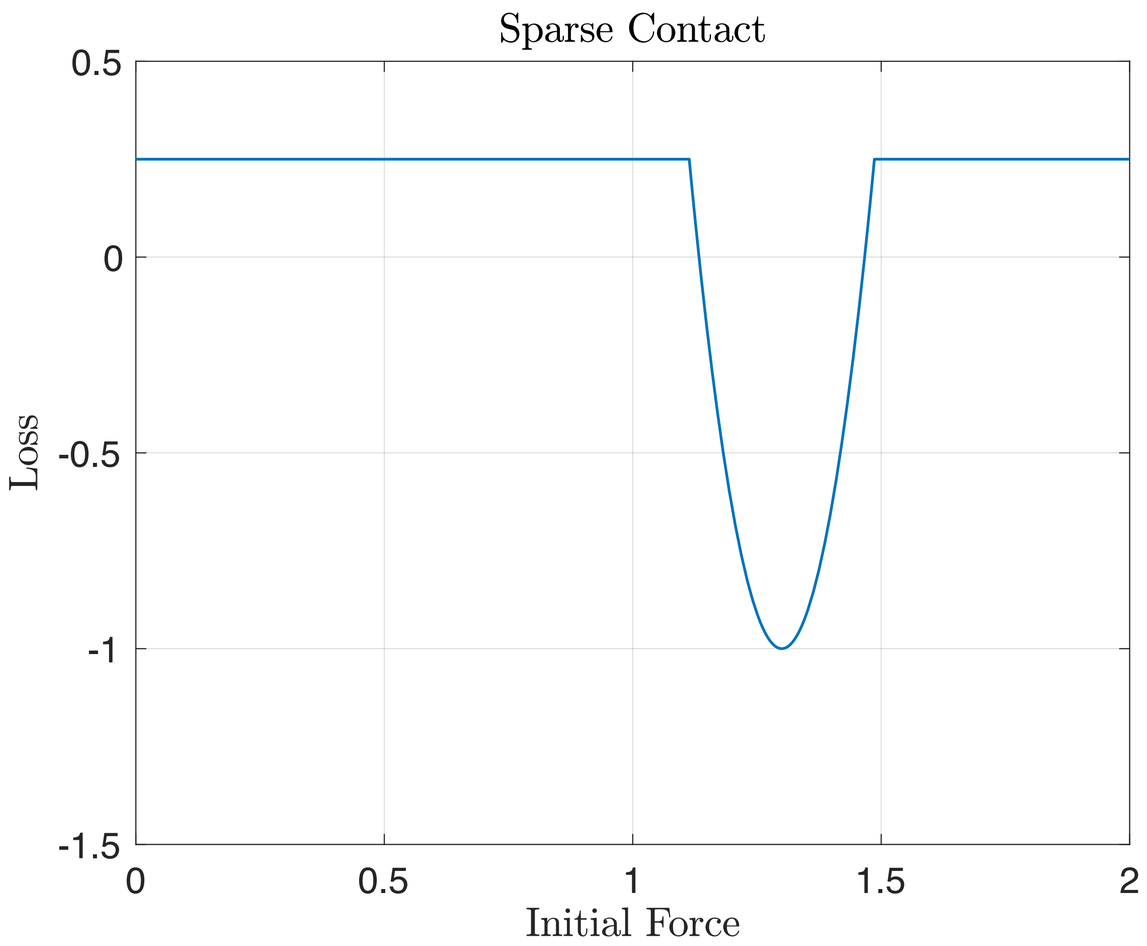




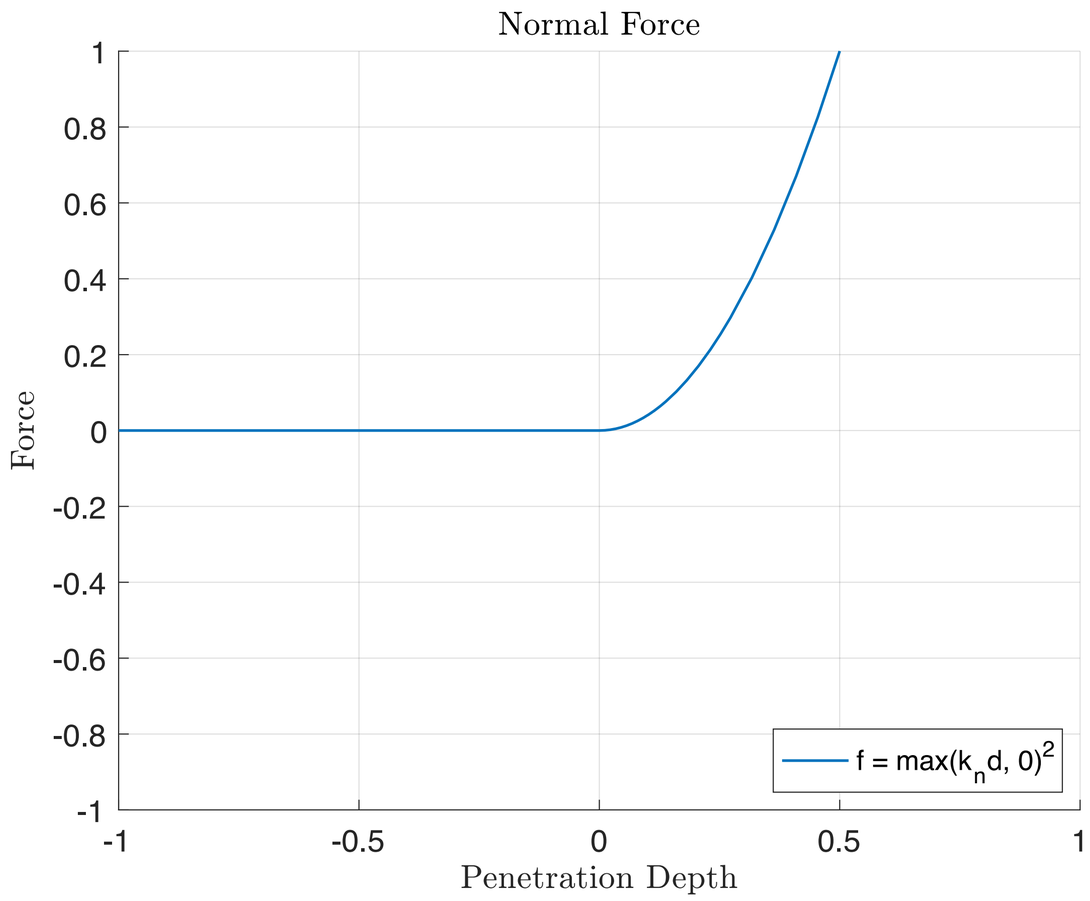
CONTACT SPARSENESS

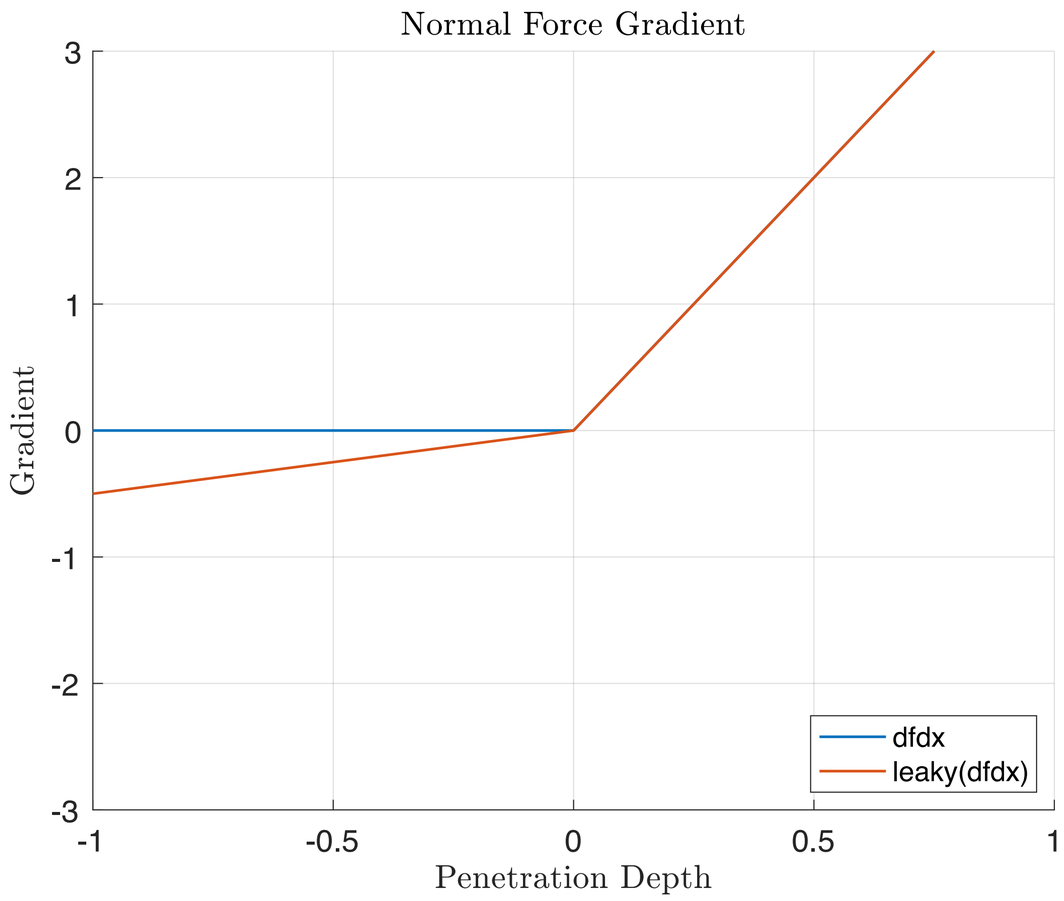
- No gradient information until contact
- Optimization stuck at local minima





CONTACT + LEAKY GRADIENTS



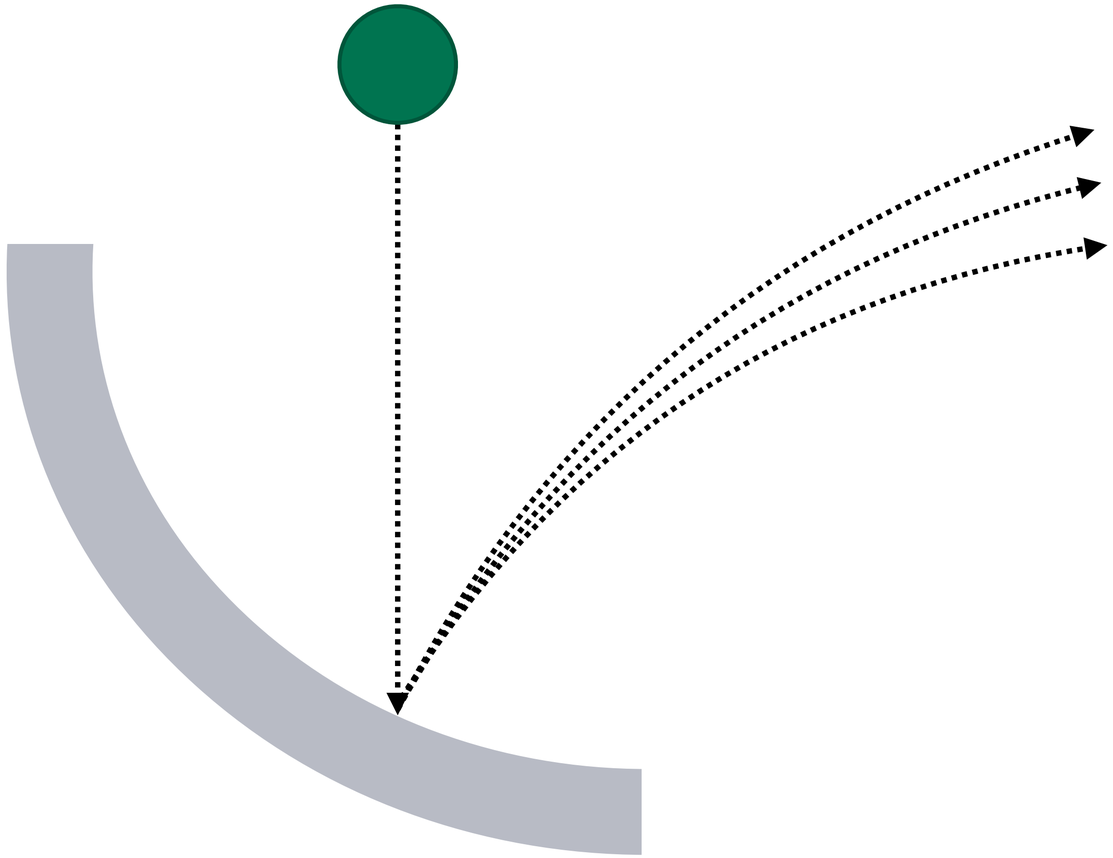


CONTACT GENERATION

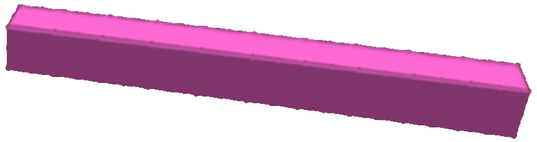
- Polygons -> contact points + normals
- Computational geometry problem
- Not autodiff friendly
- SDFs are promising
- e.g.: adjoint of SDF normal:

$$n(\mathbf{x}) = \nabla\phi(\mathbf{x})$$
$$n^*(\mathbf{x}) = \nabla^2\phi(\mathbf{x})s^*$$

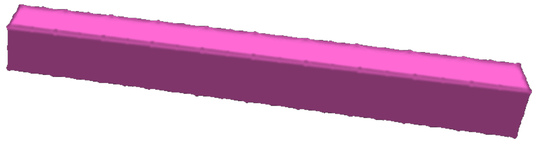
- I3D paper on SDFs accepted



PARAMETER ESTIMATION FROM VIDEO



Ground Truth

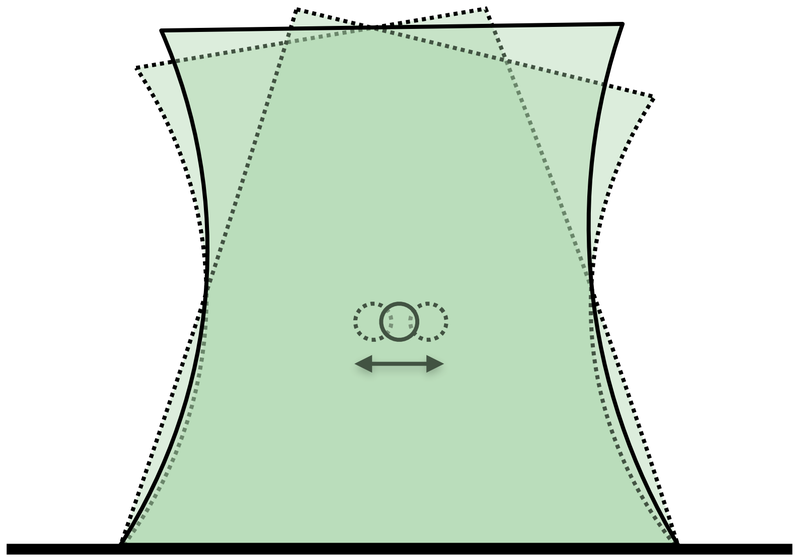


Initial Guess

After Optimization

LOCAL MINIMA

- Gradient methods are sensitive to local minima
- e.g.: oscillations of elastic bodies
- Momentum methods like Nesterov can ‘jump’ over these to some degree
- Combine stochastic exploration with gradient-based optimization



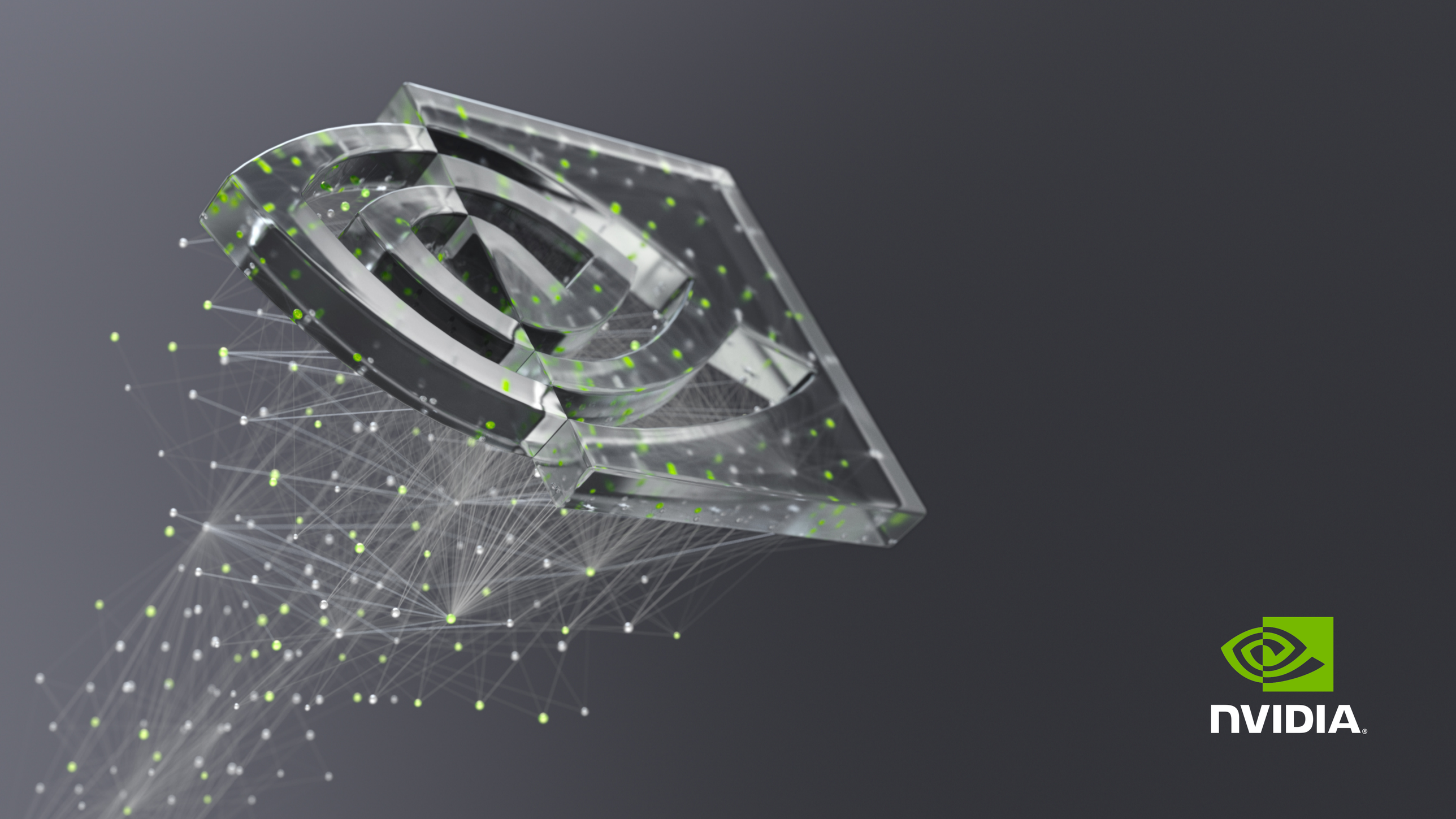
FUTURE WORK

- Differentiable CUDA / HLSL compiler
- Optimization through contact
- Application to reinforcement learning
- Extension to more physical models

REFERENCES

- GradSim ICLR:
  - Paper: https://openreview.net/forum?id=c_E8kFWfhp0
- Jos Stam's SIGGRAPH talk:
  - Video: https://developer.nvidia.com/siggraph/2019/video/sig903-vid
- Related work:

  - *Griewank, A., & Walther, A. (2008). Evaluating derivatives: Principles and techniques of algorithmic differentiation*
  - *Margossian (2019). A Review of Automatic Differentiation and its Efficient Implementation*
  - *McNamara et al. (2004). Fluid Control Using the Adjoint Method*
  - *Wojtan et al. (2006). Keyframe Control of Complex Particle Systems Using the Adjoint Method*
  - *Hu et al. (2020). DiffTaichi: Differentiable Programming for Physical Simulation*


NVIDIA