Lecture 4

Resolving Particle Collisions

FUNDAMENTALS OF COMPUTER GRAPHICS

Animation & Simulation

Stanford CS248B, Fall 2022

PROFS. KAREN LIU & DOUG JAMES

Announcements

- HW1 due today
- P1: Pinball out today
  - Due in two weeks

Stanford CS 248B Fall 2022 – Programming Assignment #1

Pinball!

Colliding Particles for Fun

Summary

In your first programming assignment you will design, model, simulate and play a virtual game of pinball. The main challenge is to simulate the ball motion subject to collisions with an environment composed of rigid obstacles (mostly static but some moving). The environment will be modeled using simple 2D shape primitives, for which we can use signed-distance fields (SDFs) to help process collisions robustly, even for fast-moving balls. Your implementation will be done in Open Processing (OP), and you and your classmates will be able to share and play each others' games (without sharing the code). Yes, this will be a groovy start to CS248B.

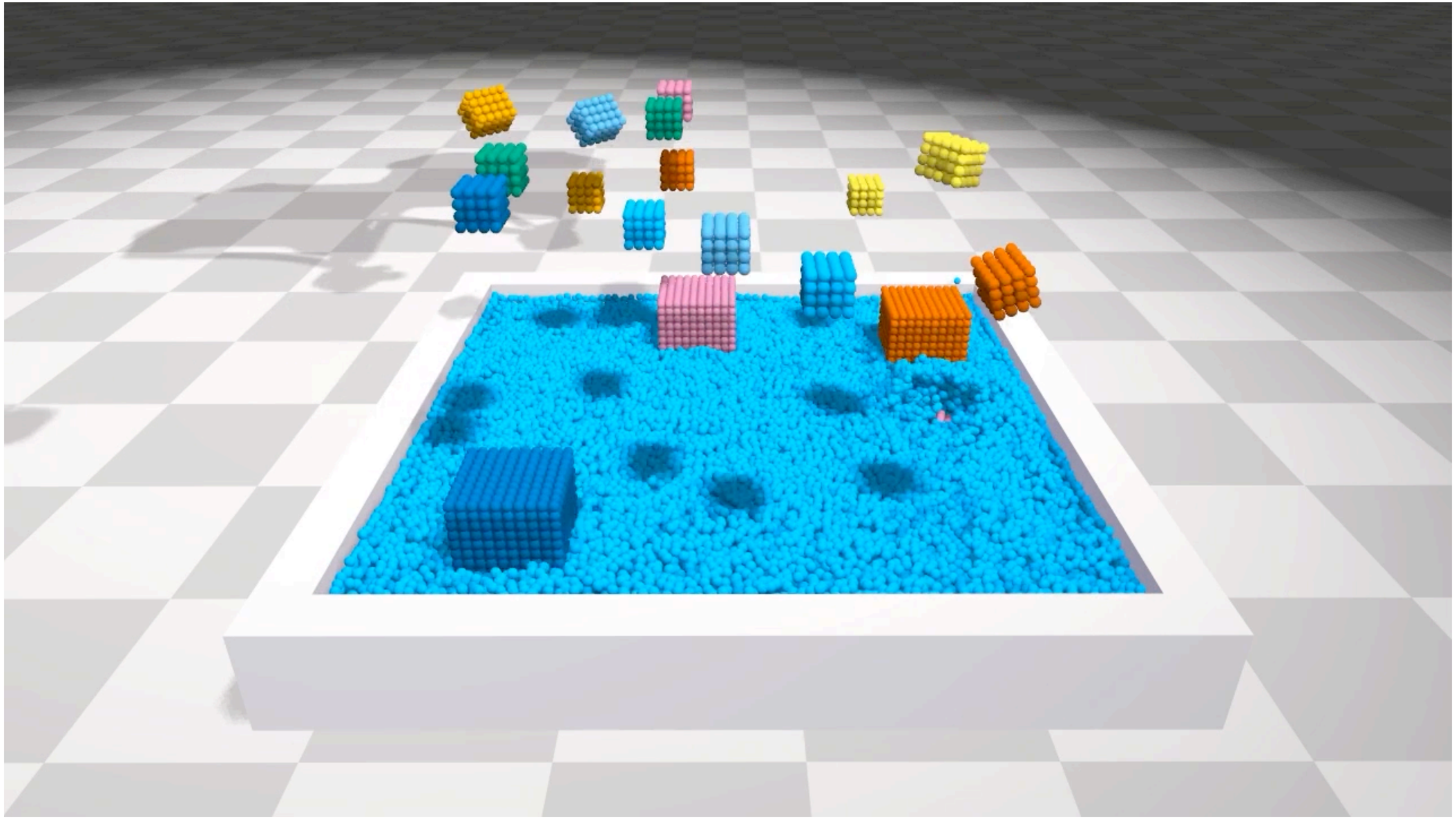
https://en.wikipedia.org/wiki/Pinball

2D Particle Simulation

For this first assignment, we will model the ball using a simple 2D point-like circular primitive (with unit mass, m=1, radius r, position **p**, and velocity **v**) falling under a small gravitational acceleration, **g**. This assumption ignores the more complex dynamics of a 3D spherical balling with rolling and frictional contact. The main challenge is not the time-stepping of the free-flight motion, but the detection and robust processing of ball-object collisions. As discussed in class, you can use a Symplectic Euler integrator for the ball dynamics, and use impulses to modify the velocity to resolve collisions. In difficult cases, such as for fast-moving balls, you will need to perform continuous collision processing to avoid missing collisions or having objects interpenetrate (more on that later).

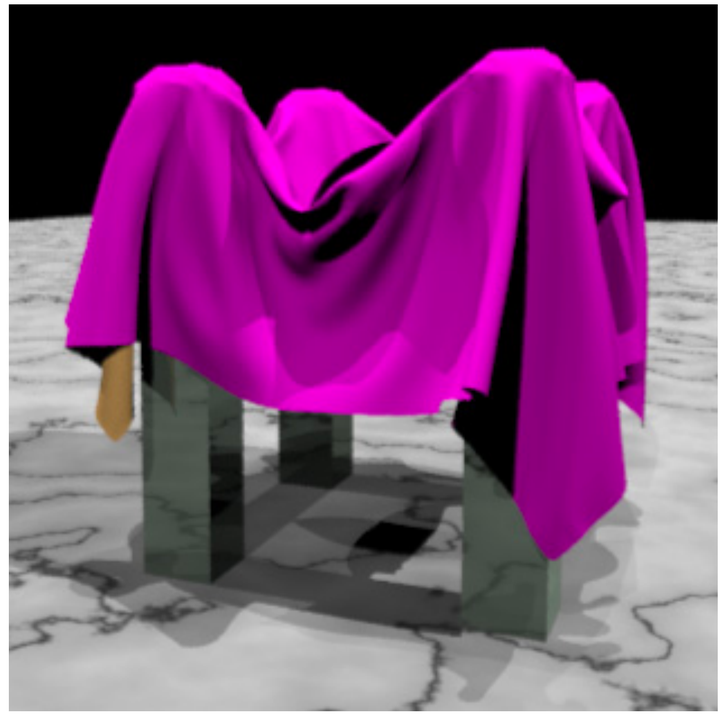
The starter code includes a `Ball class` that can `draw()` a circle, and has a `timestep(dt)` method that performs discrete collision detection using the scene's signed-distance field, and applies a suitable collision impulse. You will need to modify the time-stepping scheme to handle collisions more robustly, as discussed below.

Today: Particle Collisions



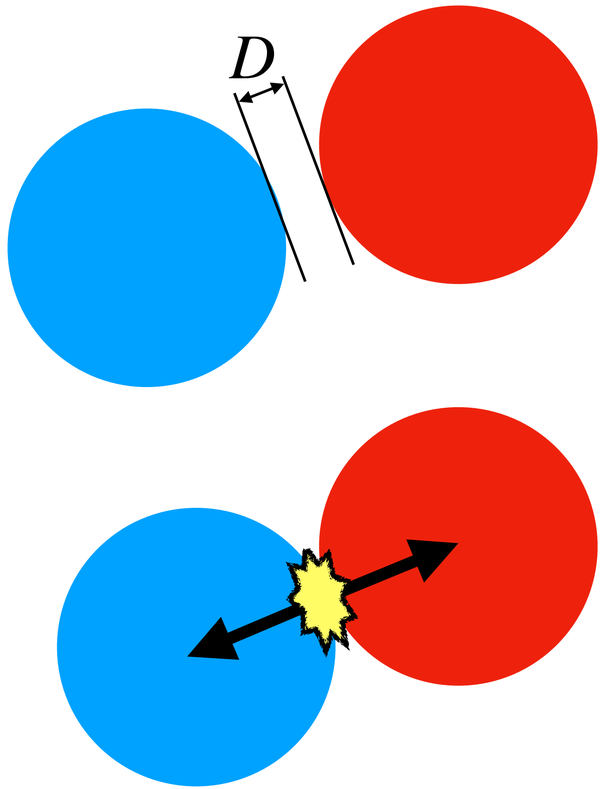
Overview

- Collision detection vs collision response
- Particle-based contact models
  - Penalty forces
  - Collision impulses. Coefficient of restitution (COR)
  - Time integration
- Proximity queries using signed-distance fields (SDFs)
  - 2D examples
  - Distances & normals
  - Transformations
  - Boolean combinations
- P1: Pinball!



[Bridson et al. 2002]

Collision detection vs collision response



- **Collision detection**
  - Geometric computation
  - Determines when and where objects collide
    - Discrete vs continuous temporal checks
- **Collision response**
  - Dynamics computation
  - Changes object velocities to respond to contact

Contact models

- Gradual vs instantaneous velocity change
  - Gradual: forces spread over multiple steps
  - Instantaneous: impulses at time of collision
    - Impulse are momentum (=force*time=mass*velocity)
- Normal and tangential components
  - Non-penetration (normal)
  - Friction (tangential)

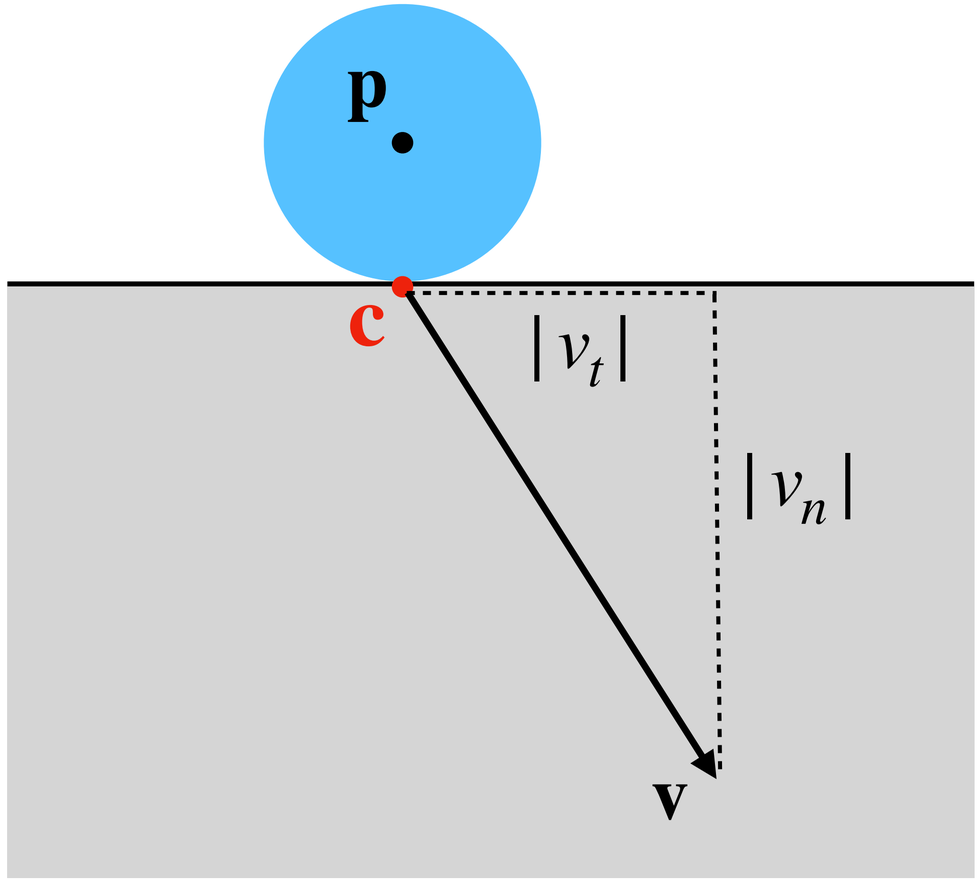
Particle-plane contact geometry (2D)

- Particle position, $\mathbf{p}$
- Particle velocity, $\mathbf{v}$
- Contact point, $\mathbf{c}$
  - Point vs point-like particle
- Unit contact normal, $\hat{\mathbf{n}}$
- Normal velocity component,

  $v_n = \mathbf{v} \cdot \hat{\mathbf{n}}$

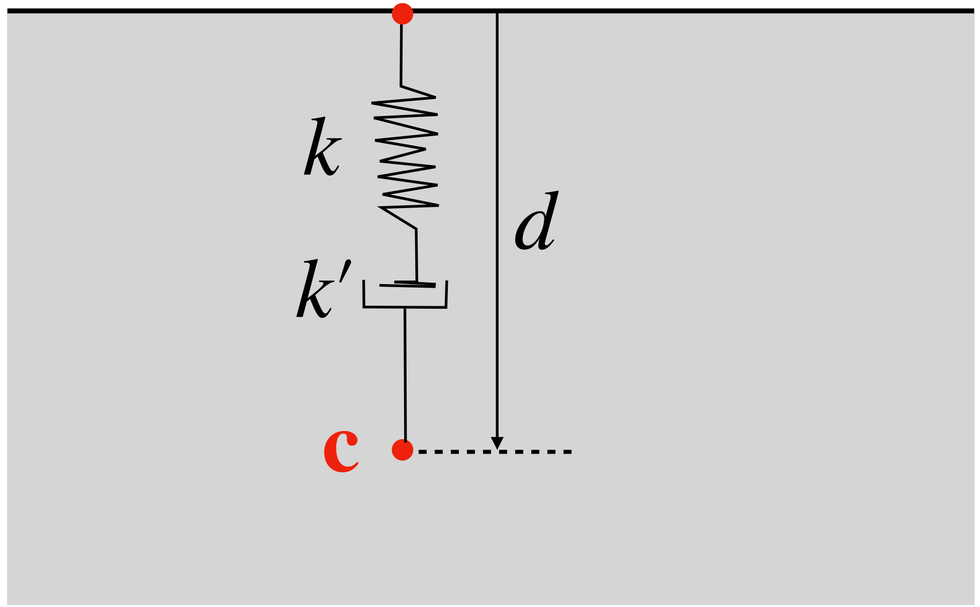
  - $v_n < 0$: Penetrating
  - $v_n > 0$: Separating
- Tangential velocity component (2D)

  $v_t = \mathbf{v} \cdot \hat{\mathbf{n}}_\perp$



Penalty contact model

- Spring-based contact models
- Penetration vs barrier
- Signed penetration depth, d
  - $d < 0$: Interpenetrating
  - $d > 0$: Out of contact
- Simplest Case: Damped linear spring
  - Normal force: $f_n = -k\,d - k'\,\dot{d}$
    - $\mathbf{f} = f_n\,\hat{\mathbf{n}}$
- Practical challenges:
  - Tuning parameters
  - Numerically stiff forces for ODE integrator $\longrightarrow$ small time-steps sizes



Impulse-based contact model

- Instantaneous velocity-level contact resolution
- Given a collision event with an immovable object and $v_n < 0$, modify the normal velocity:

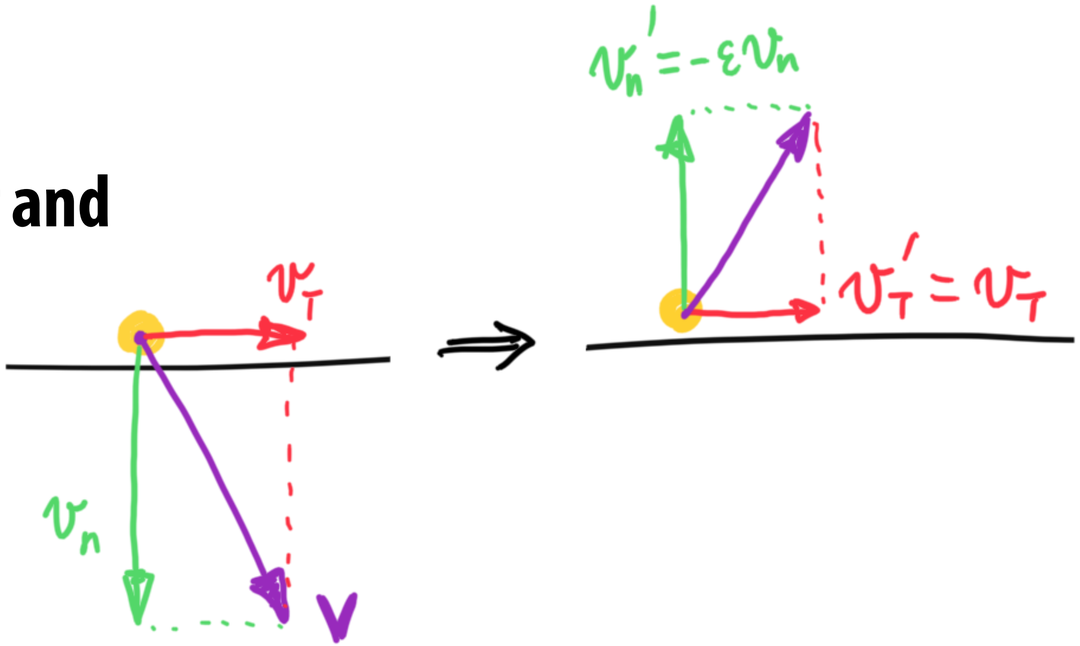
$$v_n' = -\varepsilon v_n$$

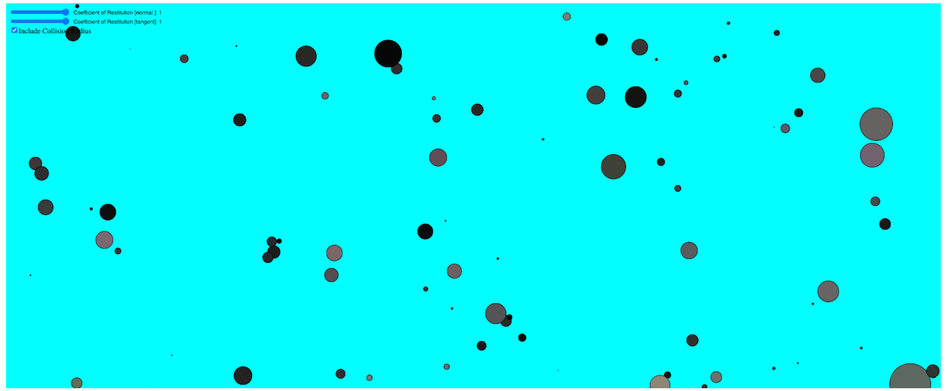
where the **coefficient of restitution** (COR), $\varepsilon \in [0,1]$, controls bounciness:

  - $\varepsilon = 1$: Perfectly elastic bounce
  - $\varepsilon = 0$: Perfectly inelastic (no bounce)

- Tangential COR models a frictional impulse:

$$v_t' = -\varepsilon_t v_t$$





Coefficient of Restitution Sketch

https://www.openprocessing.org/sketch/966506

Symplectic Euler integrator

1. **Accumulate forces:** $\mathbf{f}$ (springs, gravity, drag, etc.)
2. **Evaluate accelerations:** $\mathbf{a} = \mathbf{M}^{-1}\mathbf{f}$
3. **Timestep velocities:** $\mathbf{v} \mathrel{+}= \Delta t\ \mathbf{a}$
4. **Timestep positions:** $\mathbf{p} \mathrel{+}= \Delta t\ \mathbf{v}$

Symplectic Euler integrator with filters

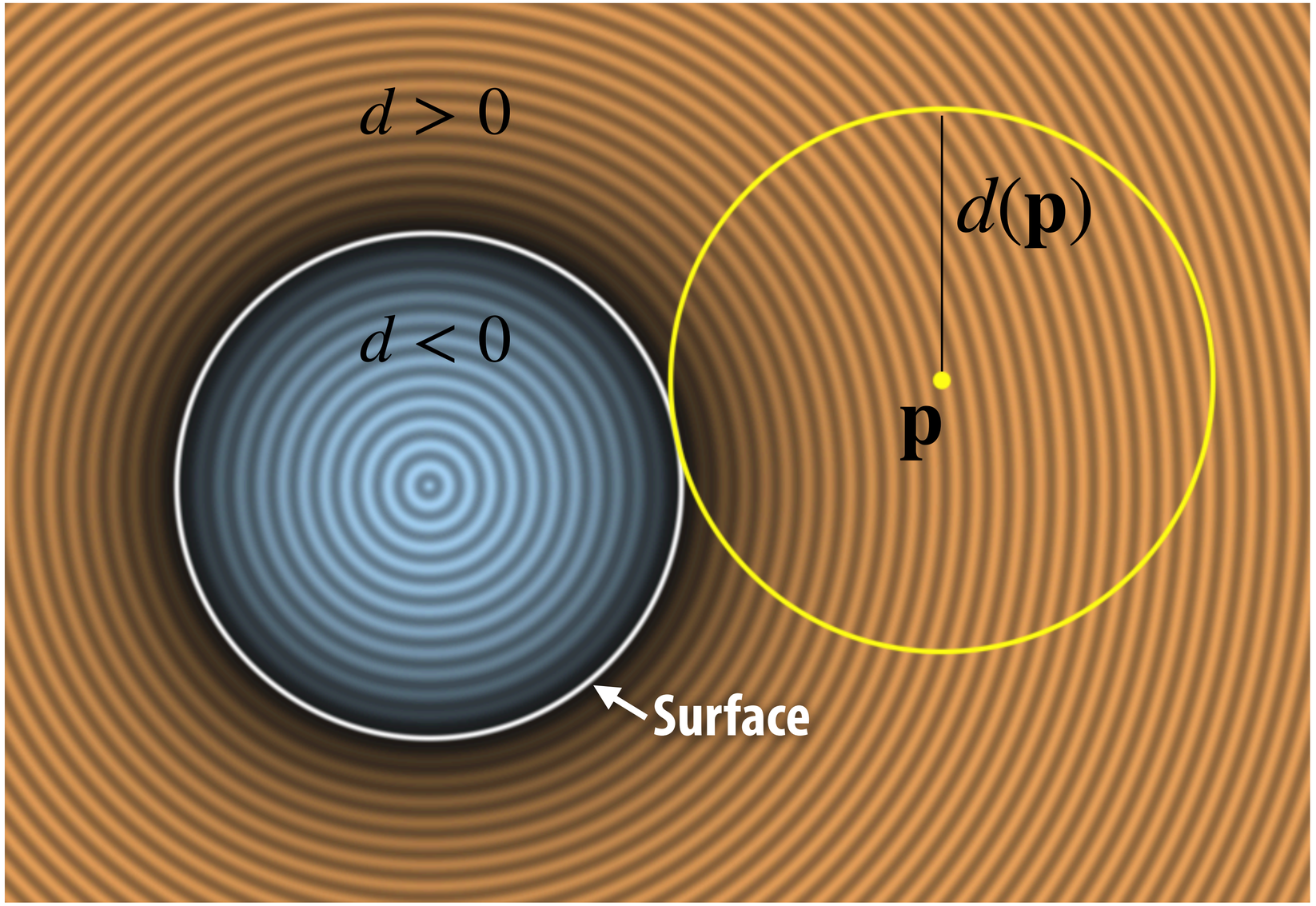
1. **Accumulate forces:** $\mathbf{f}$ (springs, gravity, drag, etc.)
2. **Evaluate accelerations:** $\mathbf{a} = \mathbf{M}^{-1}\mathbf{f}$
   - Optional: Filter accelerations for pinned particles
3. **Timestep velocities:** $\mathbf{v} \mathrel{+}= \Delta t\ \mathbf{a}$
   - Optional: Filter velocities for collisions and constraints
4. **Timestep positions:** $\mathbf{p} \mathrel{+}= \Delta t\ \mathbf{v}$
   - Optional: Filter positions

Proximity Queries using Signed-Distance Fields (SDFs)

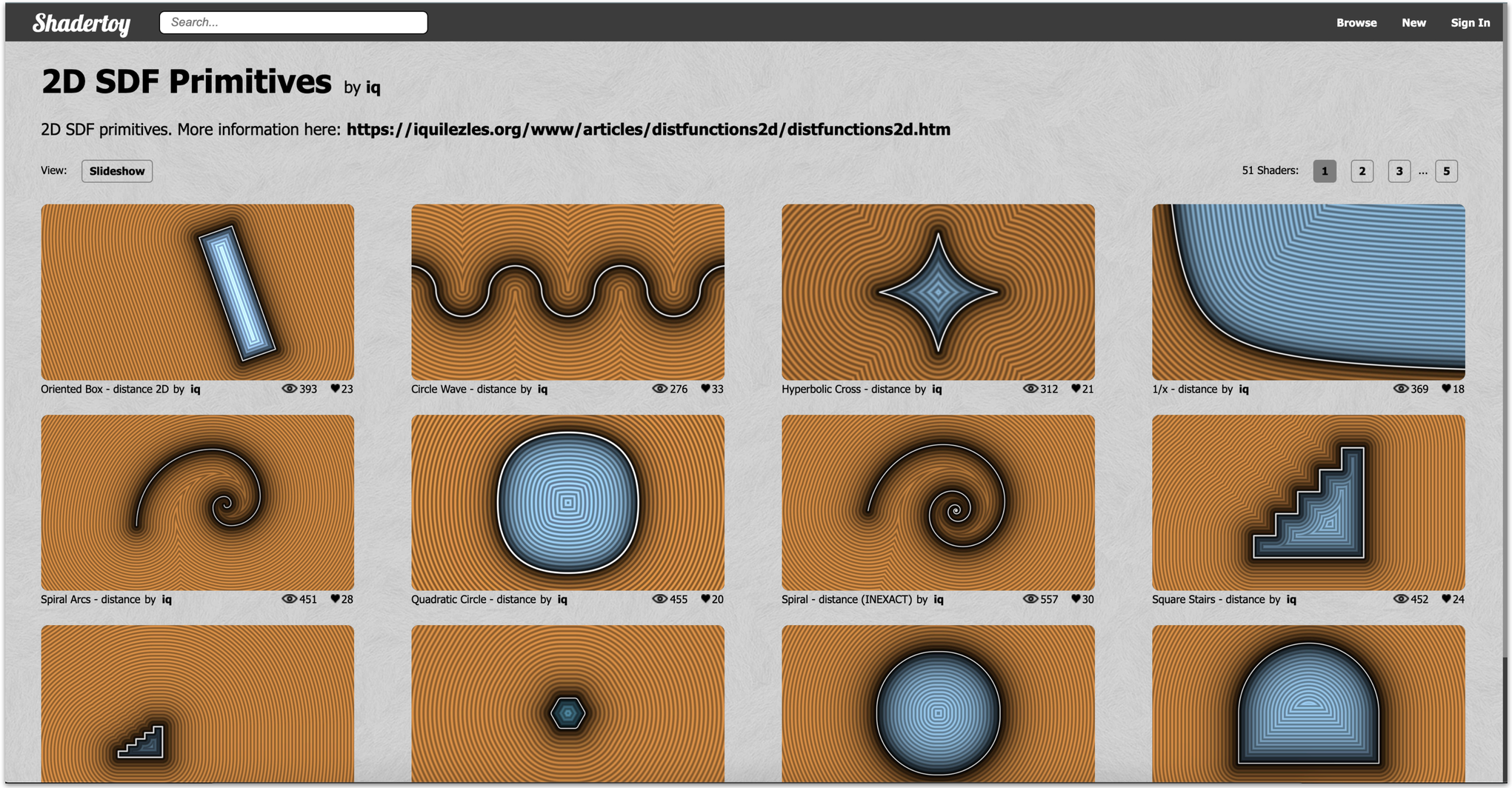
Signed-Distance Fields (SDFs)

- SDF: $d(\mathbf{p})$ is the signed distance to a closed surface
- $d(\mathbf{p}) > 0$: Outside
- $d(\mathbf{p}) = 0$: On surface
- $d(\mathbf{p}) < 0$: Inside

- Unsigned distance field: positive inside & outside



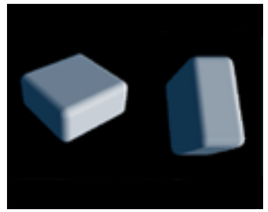
2D signed-distance field (SDF) primitives

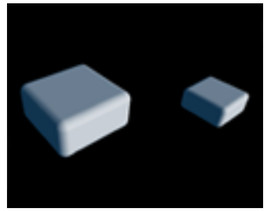


https://iquilezles.org/articles/distfunctions2d

Transforming SDF primitives

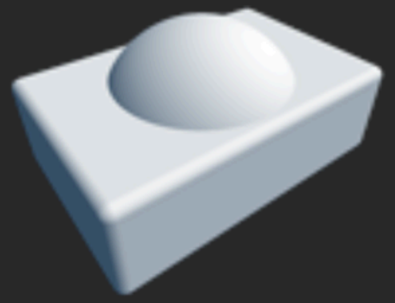
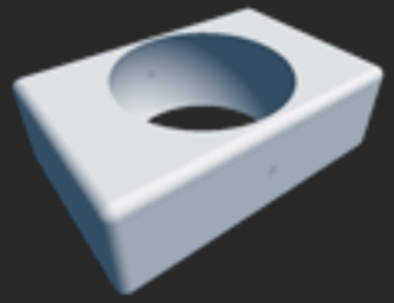
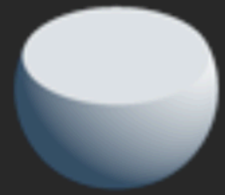
- Given SDF primitive, $d(\mathbf{p})$
- Idea: Apply the inverse transformation to $\mathbf{p}$ first
- Transformations:
  - Translate by $\mathbf{w}$: $d(\mathbf{p} - \mathbf{w})$
  - Rotate by $\mathbf{R}$: $d(\mathbf{R}^T\mathbf{p})$
  - Scale by s: $d\left(\frac{\mathbf{p}}{s}\right) s$





https://iquilezles.org/articles/distfunctions

Boolean operations on SDF shapes

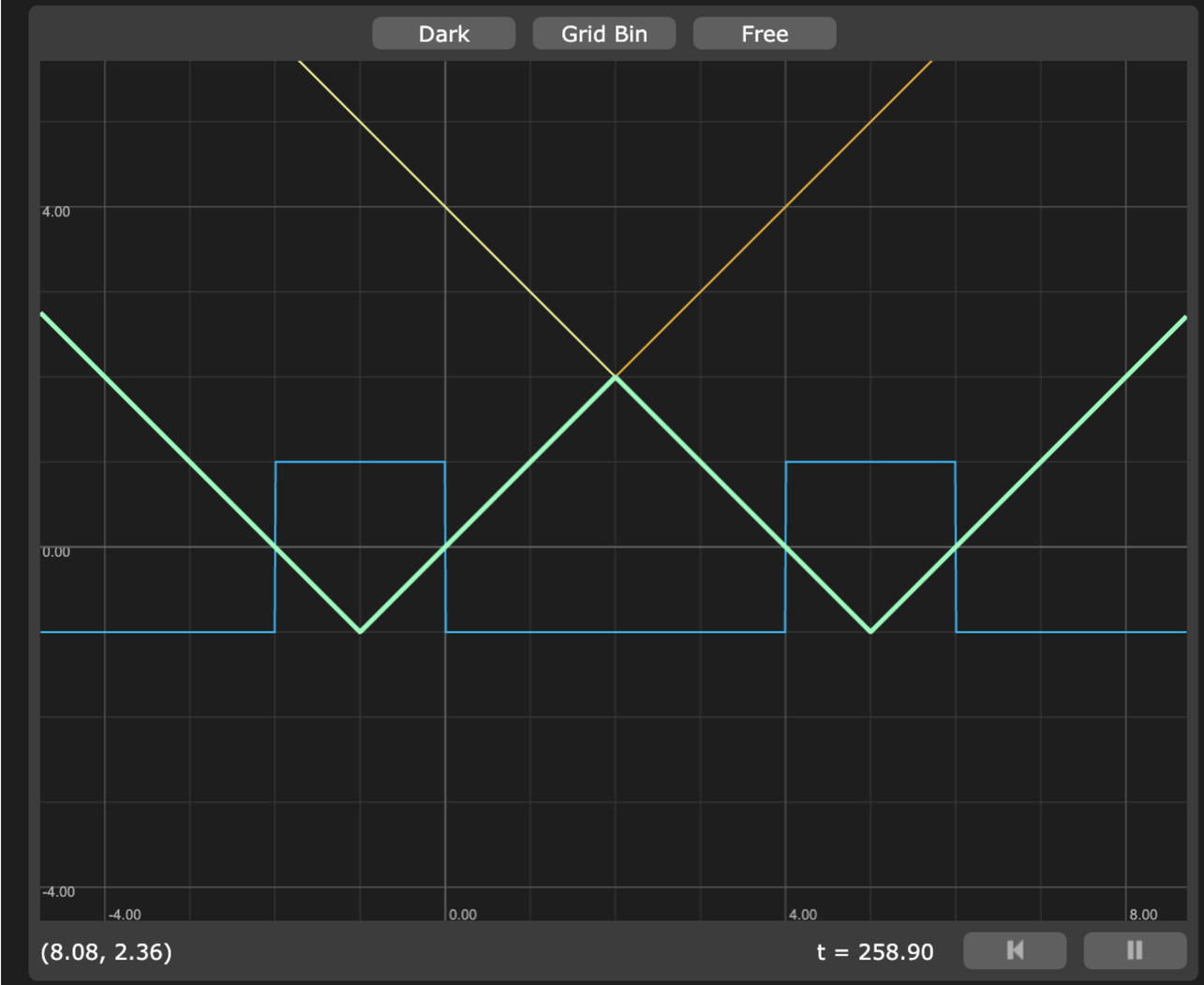
```
float opUnion( float d1, float d2 ) { return min(d1,d2); }

float opSubtraction( float d1, float d2 ) { return max(-d1,d2); }

float opIntersection( float d1, float d2 ) { return max(d1,d2); }
```

Warning: Only the union (min) operation creates valid *exterior* distance fields.

https://iquilezles.org/articles/distfunctions

Boolean Experiments in GraphToy https://graphtoy.com

Graphtoy v0.4 by Inigo Quilez (feedback from Rafæl Couto, Florian Mosleh, Nicholas Ralabate, Rich Eakin and Jason Tully). If you find Graphtoy useful, please consider supporting it by donating through my Patreon or PayPal.



Create Link for Sharing | Clear | Example 1 | Example 2 | Example 3

$f_1(x,t)$ = abs(x+1)-1

$f_2(x,t)$ = abs(x-5)-1

$f_3(x,t)$ = min(f1(x),f2(x))

$f_4(x,t)$ = -sign(f3(x))

$f_5(x,t)$ =

$f_6(x,t)$ =

()	+	-	^	**	pow(x,y)
*	/	rcp(x)	exp(x)	exp2(x)	exp10(x)
fma(x,y,z)	%	mod(x,y)	log(x)	log2(x)	log10(x)
sqrt(x)	cbrt(x)	rsqrt(x)	cos(x)	sin(x)	tan(x)
rcbrt(x)	inversesqrt(x)		acos(x)	asin(x)	atan(x)
abs(x)	sign(x)	ssign(x)	atan2(x,y)	radians(x)	degrees(x)
cosh(x)	sinh(x)	tanh(x)	ceil(x)	floor(x)	trunc(x)
acosh(x)	asinh(x)	atanh(x)	round(x)	frac(x)	fract(x)
min(x,y)	max(x,y)	saturate(x)	remap(a,b,x,c,d)		mix(a,b,x)
clamp(x,c,d)		step(a,x)	lerp(a,b,x)	tri(a,x)	sqr(a,x)

Programming Assignment #1

Pinball

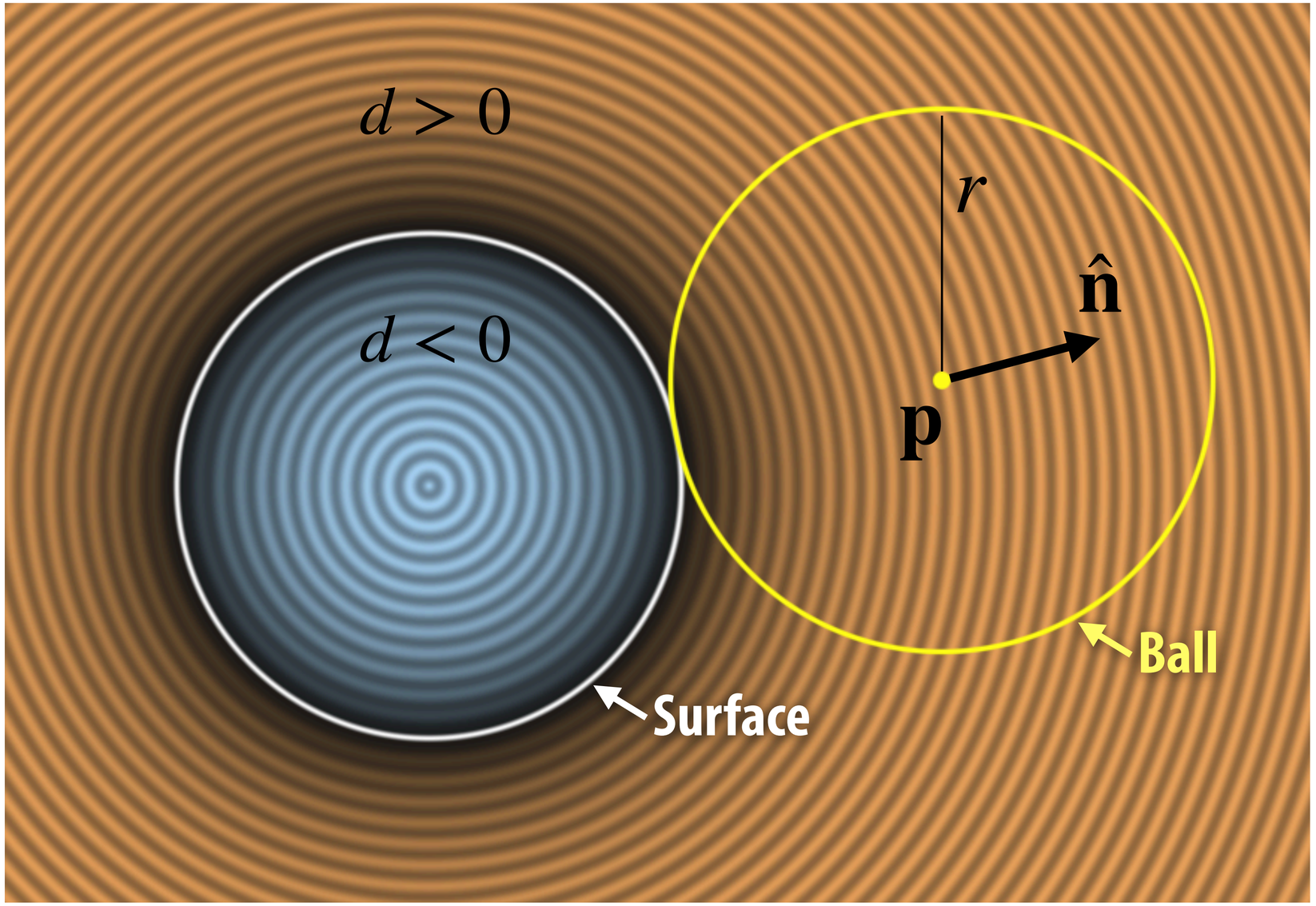
Ball collision using an SDF

- Ball radius, r
- Ball position, $\mathbf{p}$
- SDF: $d(\mathbf{p})$
- Distance of ball to object
  - $D(\mathbf{p}) = d(\mathbf{p}) - r$
  - Collision if $D(\mathbf{p}) = 0$
- Contact normal given by

$$\hat{\mathbf{n}} = \frac{\nabla d(\mathbf{p})}{\|\nabla d(\mathbf{p})\|}$$

- Contact point located at

$$\mathbf{c} = \mathbf{p} - r\,\hat{\mathbf{n}}$$

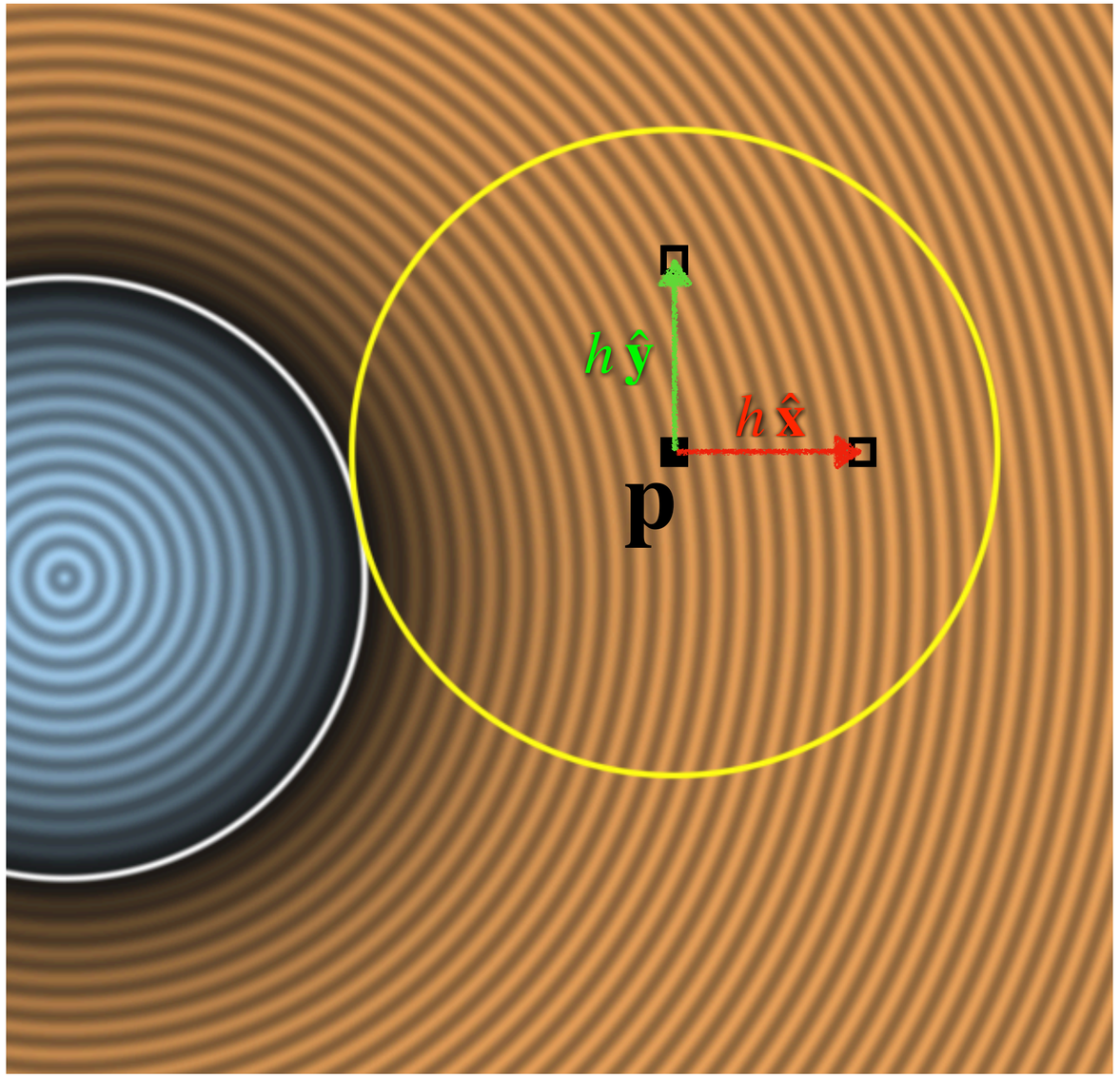


SDF Gradients

- **Option 1: Analytical gradients**
  - Code returns both $d(\mathbf{p})$ and $\nabla d(\mathbf{p})$
  - More complex code for gradients
- **Option 2: Finite difference gradients**
  - Uses code for any $d(\mathbf{p})$
  - Example: One-sided finite differences

$$\nabla d(\mathbf{p}) \approx \begin{pmatrix} \dfrac{d(\mathbf{p} + h\,\hat{\mathbf{x}}) - d(\mathbf{p})}{h} \\ \dfrac{d(\mathbf{p} + h\,\hat{\mathbf{y}}) - d(\mathbf{p})}{h} \end{pmatrix}$$

  - $O(h)$ accurate :/



Programming Assignment #1 (2 weeks)

Pinball!

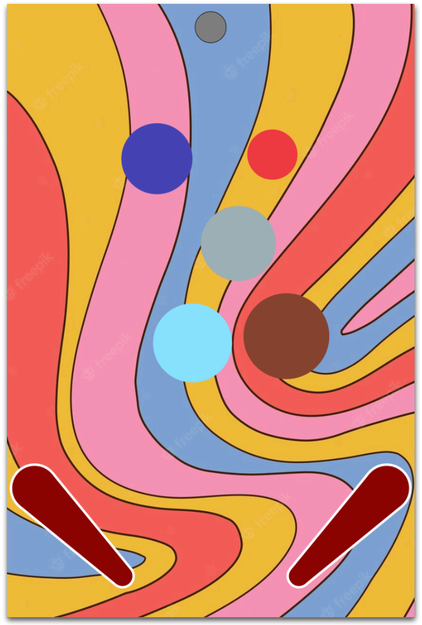
- **Document:**

  https://docs.google.com/document/d/1i3sofnG_dhPMgHVh-lXHdPe-eFs6ts85KE9woLUECMo/edit?usp=sharing

- **248B Pinball Startercode**

  https://openprocessing.org/sketch/1651988

- **Have fun!!**



Stanford CS 248B Fall 2022 – Programming Assignment #1

Pinball!

Colliding Particles for Fun

Summary

In your first programming assignment you will design, model, simulate and play a virtual game of pinball. The main challenge is to simulate the ball motion subject to collisions with an environment composed of rigid obstacles (mostly static but some moving). The environment will be modeled using simple 2D shape primitives, for which we can use signed-distance fields (SDFs) to help process collisions robustly, even for fast-moving balls. Your implementation will be done in Open Processing (OP), and you and your classmates will be able to share and play each others' games (without sharing the code). Yes, this will be a groovy start to CS248B.

https://en.wikipedia.org/wiki/Pinball

2D Particle Simulation

For this first assignment, we will model the ball using a simple 2D point-like circular primitive (with unit mass, m=1, radius r, position **p**, and velocity **v**) falling under a small gravitational acceleration, **g**. This assumption ignores the more complex dynamics of a 3D spherical balling with rolling and frictional contact. The main challenge is not the time-stepping of the free-flight motion, but the detection and robust processing of ball-object collisions. As discussed in class, you can use a Symplectic Euler integrator for the ball dynamics, and use impulses to modify the velocity to resolve collisions. In difficult cases, such as for fast-moving balls, you will need to perform continuous collision processing to avoid missing collisions or having objects interpenetrate (more on that later).

The starter code includes a `Ball class` that can `draw()` a circle, and has a `timestep(dt)` method that performs discrete collision detection using the scene's signed-distance field, and applies a suitable collision impulse. You will need to modify the time-stepping scheme to handle collisions more robustly, as discussed below.