太极图形课

第01讲 Hello world: basis in Taichi

09/22/2021

taichi

太极图形课

第01讲 Hello world: basis in Taichi

09/22/2021

taichi

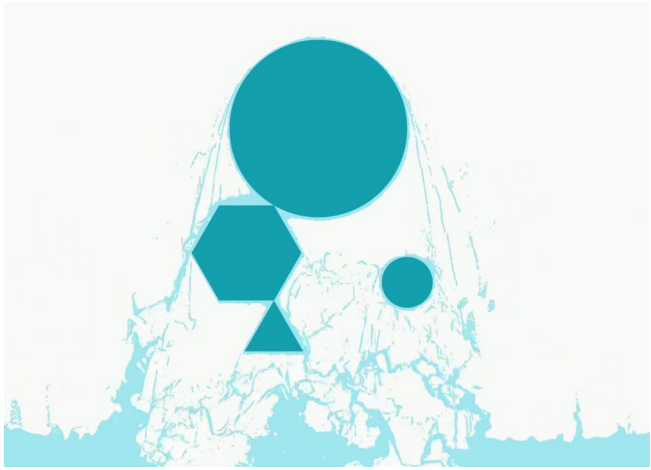
taichi

太极图形祝您中秋快乐

Happy Mid-Autumn Festival

Recap

- OP / ED Animation
  - Courtesy of Andrew Sun



- Taichi installation
  - Local build
  - Taichi zoo

Recap

- OP / ED Animation
  - Courtesy of Andrew Sun

- Taichi installation
  - Local build
  - Taichi zoo

python3 –m pip install taichi

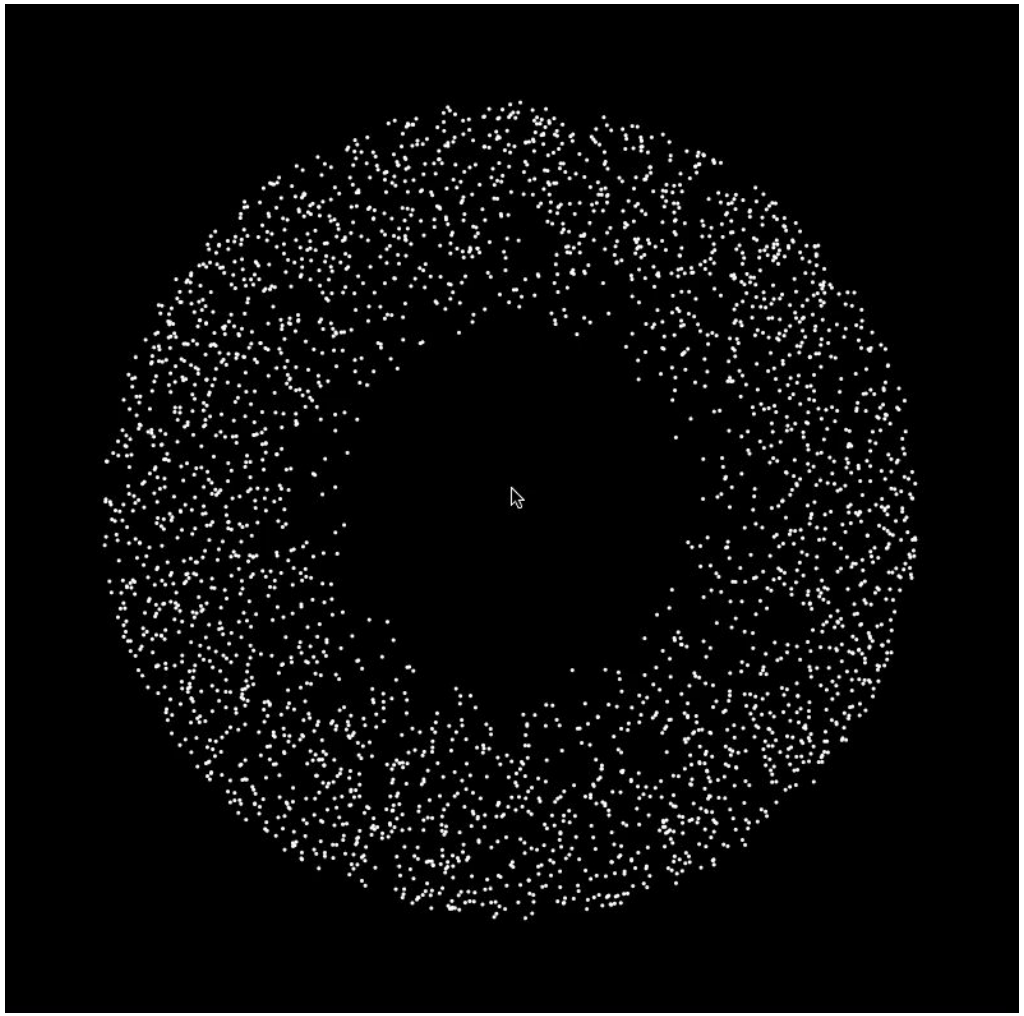
Announcing a new teaching assistant

- Mingrui Zhang 张铭睿（https://github.com/erizmr）

Hello World!

```
def Hello():
    print("Hello World!")

Hello()
```

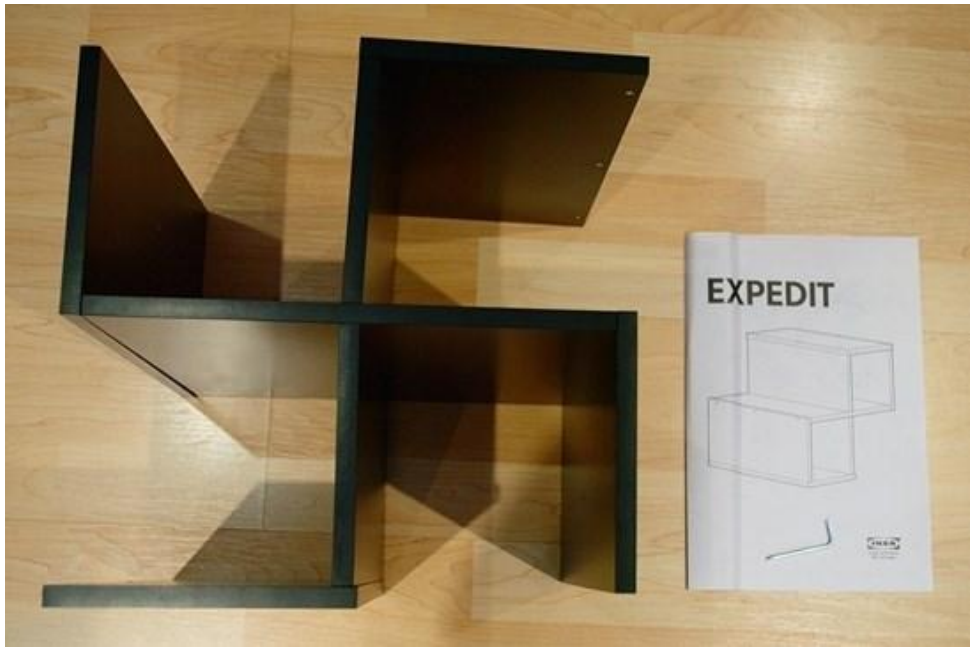
Hello World @ Taichi



```python
import taichi as ti

ti.init(ti.gpu)

# gravitational constant 6.67408e-11, using 1 for simplicity
G = 1
PI = 3.141592653

# number of planets
N = 300
# unit mass
m = 5
# galaxy size
galaxy_size = 0.4
# planet radius (for rendering)
planet_radius = 2
# init vel
init_vel = 120

# time-step size
h = 1e-5
# substepping
substepping = 10

# pos, vel and force of the planets
# Nx2 vectors
pos = ti.Vector.field(2, ti.f32, N)
vel = ti.Vector.field(2, ti.f32, N)
force = ti.Vector.field(2, ti.f32, N)

@ti.kernel
def initialize():
    center = ti.Vector([0.5, 0.5])
    for i in range(N):
        theta = ti.random() * 2 * PI
        r = (ti.sqrt(ti.random()) * 0.7 + 0.3) * galaxy_size
        offset = r * ti.Vector([ti.cos(theta), ti.sin(theta)])
        pos[i] = center+offset
        vel[i] = [-offset.y, offset.x]
        vel[i] *= init_vel

@ti.kernel
def compute_force():
    # clear force
    for i in range(N):
        force[i] = ti.Vector([0.0, 0.0])

    # compute gravitational force
    for i in range(N):
        p = pos[i]
        for j in range(N):
            if i != j: # double the computation for a better memory footprint and load balance
                diff = p-pos[j]
                r = diff.norm(1e-5)

                # gravitational force -(GMm / r^2) * (diff/r) for i
                f = -G * m * m * (1.0/r)**3 * diff

                # assign to each particle
                force[i] += f

@ti.kernel
def update():
    dt = h/substepping
    for i in range(N):
        #symplectic euler
        vel[i] += dt*force[i]/m
        pos[i] += dt*vel[i]

gui = ti.GUI('N-body problem', (512, 512))

initialize()
while gui.running:

    for i in range(substepping):
        compute_force()
        update()

    ...

    gui.clear(0x112F41)
    gui.circles(pos.to_numpy(), color=0xffffff, radius=planet_radius)
    gui.show()
```

The N-body problem: https://zoo.taichi.graphics/playground/e9baa99382789771692 80ba40355d302

Build your first project…



Data

Computation

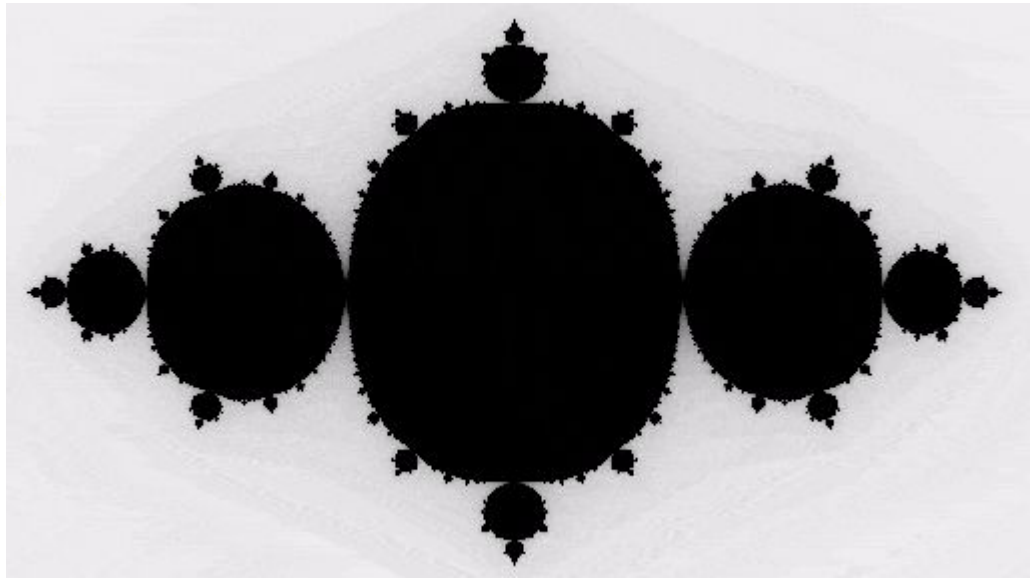
Visualization

Initialization

import taichi as ti

- `import taichi as ti`
- `ti.init(arch=ti.gpu)`

```python
import taichi as ti

ti.init(arch=ti.gpu) # Run on GPU by default

n = 320
pixels = ti.field(dtype=float, shape=(n * 2, n))

@ti.func
def complex_sqr(z):
    return ti.Vector([z[0]**2 - z[1]**2, z[1] * z[0] * 2])

@ti.kernel
def paint(t: float):
    for i, j in pixels:  # Parallized over all pixels
        c = ti.Vector([-0.8, ti.cos(t) * 0.2])
        z = ti.Vector([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

gui = ti.GUI("Julia Set", res=(n * 2, n))

for i in range(1000000):
    paint(i * 0.03)
    gui.set_image(pixels)
    gui.show()
```



ti.init()

- The entry point for ALL Taichi project
  - Most important arguments: arch = ti.cpu/ti.gpu/ti.arm/ti.x64/ti.cuda…

	CPU	CUDA	OpenGL	Apple Metal	Vulkan
Windows	YES	YES	YES	NO	WIP
Linux	YES	YES	YES	NO	YES
macOS	YES	NO	NO	YES	WIP

Taichi v.s. Python

The Python frontend of Taichi

```python
import taichi as ti

ti.init(arch=ti.cpu)

d = 1

def foo():
    d_python = d
    print("d_python =", d_python)

@ti.kernel
def bar():
    d_taichi = d
    print("d_taichi =", d_taichi)

d = d + 1 # d = 2
foo()     # d_python = 2
bar()     # d_taichi = 2
d = d + 1 # d = 3
foo()     # d_python = 3
bar()     # d_taichi = 2
```

Python-scope

Taichi-scope

Taichi-scope v.s. Python-scope

- Python-scope
  - Everything in a normal Python script is in the **Python-scope**

```
import taichi as ti

ti.init(arch=ti.cpu)

def foo():
    print("This is a normal python function")

foo()
```

Taichi-scope v.s. Python-scope

- Taichi-scope
  - Everything decorated by @ti.kernel or @ti.func is in the **Taichi-scope**

```
import taichi as ti

ti.init(arch=ti.cpu)

@ti.kernel
def foo():
    print("This is now a Taichi kernel")

foo()
```

Data

Primitive types

- signed integers: ti.i8, ti.i16, ti.i32, ti.i64
- unsigned integers: ti.u8, ti.u16, ti.u32, ti.u64
- floating points: ti.f32, ti.f64

Primitive types

- signed integers: ti.i8, ti.i16, ti.i32, ti.i64
- unsigned integers: ti.u8, ti.u16, ti.u32, ti.u64
- floating points: ti.f32, ti.f64

Type	i8	i16	i32	i64	u8	u16	u32	u64	f32	f64
CPU/CUDA	YES	YES	YES	YES	YES	YES	YES	YES	YES	YES
Metal	YES	YES	YES	NO	YES	YES	YES	NO	YES	NO
OpenGL	NO	NO	YES	EXT	NO	NO	NO	NO	YES	YES

Default types

- using int or float for default types
- can be changed via ti.init

```
ti.init(default_fp=ti.f32) # float = ti.f32
ti.init(default_fp=ti.f64) # float = ti.f64

ti.init(default_ip=ti.i32) # int = ti.i32
ti.init(default_ip=ti.i64) # int = ti.i64
```

Type promotions

- Taichi picks the more precise type to store the algebraic results
  - i32+f32=f32
  - i32+i64=i64

Type Casts

- Implicit casts
  - static types **within the Taichi scope**

```
import taichi as ti

ti.init(arch=ti.cpu)

def foo():
    a = 1
    a = 2.7
    print(a)

foo() #2.7
```

```
import taichi as ti

ti.init(arch=ti.cpu)

@ti.kernel
def foo():
    a = 1
    a = 2.7
    print(a)

foo() #2
```

Type Casts

- variable = ti.cast(variable, type)

```
import taichi as ti

ti.init(arch=ti.cpu)

@ti.kernel
def foo():
    a = 1.7
    b = ti.cast(a, ti.i32)
    c = ti.cast(b, ti.f32)
    print("b =", b) # b = 1
    print("c =", c) # c = 1.0

foo()
```

Compound types

- Using ti.types to **create** compound types including:
  - vector / matrix / struct

```
import taichi as ti

ti.init(arch=ti.cpu)

vec3f = ti.types.vector(3, ti.f32)
mat2f = ti.types.matrix(2, 2, ti.f32)
ray = ti.types.struct(ro=vec3f, rd=vec3f, l=ti.f32)

@ti.kernel
def foo():
    a = vec3f(0.0)
    print(a)          # [0.0, 0.0, 0.0]
    d = vec3f(0.0, 1.0, 0.0)
    print(d)          # [0.0, 1.0, 0.0]
    B = mat2f([[1.5, 1.4], [1.3, 1.2]])
    print("B =", B)        # B = [[1.5, 1.4], [1.3, 1.2]]
    r = ray(ro=a, rd=d, l=1)
    print("r.ro =", r.ro)  # r.ro = [0.0, 0.0, 0.0]
    print("r.rd =", r.rd)  # r.rd = [0.0, 1.0, 0.0]

foo()
```

Compound types

- Predefined keywords for compound types:
  - ti.Vector / ti.Matrix / ti.Struct

```
import taichi as ti

ti.init(arch=ti.cpu)

@ti.kernel
def foo():
    a = ti.Vector([0.0, 0.0, 0.0])
    print(a)          # [0.0, 0.0, 0.0]
    d = ti.Vector([0.0, 1.0, 0.0])
    print(d)          # [0.0, 1.0, 0.0]
    B = ti.Matrix([[1.5, 1.4], [1.3, 1.2]])
    print("B =", B)         # B = [[1.5, 1.4], [1.3, 1.2]]
    r = ti.Struct(v1=a, v2=d, l=1)
    print("r.v1 =", r.v1)  # r.v1 = [0.0, 0.0, 0.0]
    print("r.v2 =", r.v2)  # r.v2 = [0.0, 1.0, 0.0]

foo()
```

Compound types

- Access compound elements using [i,j,k,...] indexing

```python
import taichi as ti

ti.init(arch=ti.cpu)

@ti.kernel
def foo():
    a = ti.Vector([1.0, 2.0, 3.0])
    print(a[1]) # 2.0

    B = ti.Matrix([[1.5, 1.4], [1.3, 1.2]])
    print(B[1,0]) # 1.3

foo()
```

ti.field

- “a global N-d array of elements”

```
heat_field = ti.field(dtype=ti.f32, shape=(256, 256))
```

ti.field

- “a global N-d array of elements”
  - global: can be read/written from both the Taichi-scope and the Python-scope
  - N-d: (Scalar: N=0), (Vector: N=1), (Matrix: N=2), (N = 3, 4, 5, …)
  - elements: scalar, vector, matrix, struct

ti.field

- “a global N-d array of elements”
  - global: can be read/written from both the Taichi-scope and the Python-scope
  - N-d: (Scalar: N=0), (Vector: N=1), (Matrix: N=2), (N = 3, 4, 5, …)
  - elements: scalar, vector, matrix, struct

ti.field

- “a global N-d array of elements”
  - global: can be read/written from both the Taichi-scope and the Python-scope
  - N-d: (Scalar: N=0), (Vector: N=1), (Matrix: N=2), (N = 3, 4, 5, …)
  - elements: scalar, vector, matrix, struct

ti.field

- "a global N-d array of elements"
  - global: can be read/written from both the Taichi-scope and the Python-scope
  - N-d: (Scalar: N=0), (Vector: N=1), (Matrix: N=2), (N = 3, 4, 5, …)
  - elements: scalar, vector, matrix, struct
  - access elements in a field using [i,j,k,…] indexing

```
import taichi as ti

ti.init(arch=ti.cpu)

pixels = ti.field(dtype=float, shape=(16, 8))

pixels[1, 2] = 42.0
```

```
import taichi as ti

ti.init(arch=ti.cpu)

vf = ti.Vector.field(3, ti.f32, shape=4)

@ti.kernel
def foo():
    v = ti.Vector([1, 2, 3])
    vf[0] = v
```

ti.field

- “a global N-d array of elements”
  - global: can be read/written from both the Taichi-scope and the Python-scope
  - N-d: (Scalar: N=0), (Vector: N=1), (Matrix: N=2), (N = 3, 4, 5, …)
  - elements: scalar, vector, matrix, struct
  - access elements in a field using [i,j,k,…] indexing
    - Special case, access a zero-d field using [None]

```
zero_d_scalar = ti.field(ti.f32, shape=())
zero_d_scalar[None] = 1.5

zero_d_vec = ti.Vector.field(2, ti.f32, shape=())
zero_d_vec[None] = ti.Vector([2.5, 2.6])
```

ti.field examples

- “3D gravitational field in a 256x256x128 room”

```
gravitational_field = ti.Vector.field(n = 3,dtype=ti.f32,shape=(256,256,128))
```

- “2D strain-tensor field in a 64x64 grid”

```
strain_tensor_field = ti.Matrix.field(n = 2,m = 2, dtype=ti.f32, shape=(64,64))
```

- “a global scalar that I want to access in a Taichi kernel”

```
global_scalar = ti.field(dtype=ti.f32, shape=())
```

Example: N-body

```
# gravitational constant 6.67408e-11, using 1 for simplicity
G = 1
PI = 3.141592653

# number of planets
N = 300
# unit mass
m = 5
# galaxy size
galaxy_size = 0.4
# planet radius (for rendering)
planet_radius = 2
# init vel
init_vel = 120

# time-step size
h = 1e-5
# substepping
substepping = 10
```

```
# pos, vel and force of the planets
# Nx2 vectors
pos = ti.Vector.field(2, ti.f32, N)
vel = ti.Vector.field(2, ti.f32, N)
force = ti.Vector.field(2, ti.f32, N)
```

Computation

Kernels

- A Python function decorated by @ti.kernel is a Taichi kernel
  - Taichi kernels can only be called **from the Python scope**

```
import taichi as ti

ti.init(arch=ti.cpu)

@ti.kernel
def foo():
    print("foo")

@ti.kernel
def bar():
    print("bar")

foo()
bar()
```

```
import taichi as ti

ti.init(arch=ti.cpu)

def foo():
    print("foo")
    bar()

@ti.kernel
def bar():
    print("bar")

foo()
```

```
import taichi as ti

ti.init(arch=ti.cpu)

@ti.kernel
def foo():
    print("foo")
    bar()

@ti.kernel
def bar():
    print("bar")

foo()
```

For-loops in a @ti.kernel

- For loops at the **outermost scope** in a Taichi kernel is **automatically parallelized**

```
@ti.kernel
def fill():
    for i in range(10): # Parallelized
        x[i] += i

        s = 0
        for j in range(5): # Serialized in each parallel thread
            s += j

        y[i] = s

    for k in rang(20): # Parallelized
        z[k] = k
```

For-loops in a @ti.kernel

- Outermost scope ?

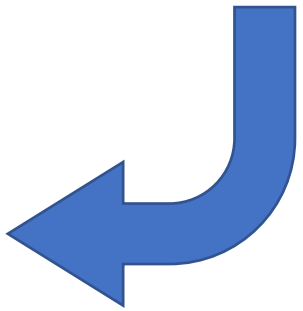
```
import taichi as ti

ti.init(arch=ti.cpu)

@ti.kernel
def foo(k: ti.i32):
    for i in range(10): # Parallelized :-)
        if k > 42:
            ...

@ti.kernel
def bar(k: ti.i32):
    if k > 42:
        for i in range(10): # Serial :-(
            ...
```

For-loops in a @ti.kernel

- Design your for loops for best performance

```
def my_for_loop():
    for i in range(10): # I don't want to parallelize this for
        for j in range(100): # I want to parallelize this for
            ...

my_for_loop()
```

```
def my_for_loop():
    for i in range(10):
        my_taichi_for()

@ti.kernel
def my_taichi_for():
    for j in range(100):
        ...

my_for_loop()
```



For-loops in a @ti.kernel

- break is NOT supported in the parallel for-loops

```
@ti.kernel
def foo():
  for i in range(10):
      ...
      break # Error!

@ti.kernel
def foo():
  for i in range(10):
      for j in range(10):
          ...
          break # OK!
```

For-loops in a @ti.kernel

- Race condition
  - Taichi uses += as an atomic add
  - The compiler optimizes for unnecessary atomic operations

```
@ti.kernel
def sum():
    for i in range(10):
        # 1. OK
        total[None] += x[i]

        # 2. OK
        ti.atomic_add(total[None], x[i])

        # 3. data race
        total[None] = total[None] + x[i]
```

For-loops in a @ti.kernel

- Types of for-loops in Taichi
  - range-for: loops over a **range**, identical to Python range-for
  - struct-for: loops over a **ti.field**, only lives at the outermost scope

```
import taichi as ti

ti.init(arch=ti.cpu)

N = 10
x = ti.field(dtype=ti.i32, shape=N)

@ti.kernel
def foo():
  for i in range(N):
      x[i] = i

foo()
```

```
import taichi as ti

ti.init(arch=ti.cpu)

N = 10
x = ti.Vector.field(2,dtype=ti.i32, shape=(N,N))

@ti.kernel
def foo():
  for i,j in x:
      x[i,j] = ti.Vector([i, j])

foo()
```

For-loops in a @ti.kernel

- Types of for-loops in Taichi
  - range-for: loops over a **range**, identical to Python range-for
  - struct-for: loops over a **ti.field**, only lives at the outermost scope

```
import taichi as ti

ti.init(arch=ti.cpu)

N = 10
x = ti.field(dtype=ti.i32, shape=N)

@ti.kernel
def foo():
  for i in range(N):
      x[i] = i

foo()
```

```
import taichi as ti

ti.init(arch=ti.cpu)

N = 10
x = ti.Vector.field(2,dtype=ti.i32, shape=(N,N))

@ti.kernel
def foo():
  for i,j in x:
      x[i,j] = ti.Vector([i, j])

foo()
```

Kernel arguments

- At most 8 parameters
- Pass from the Python scope to the Taichi scope
- Must be type-hinted
- Scalar Only
- Pass by value

Kernel arguments

- At most 8 parameters
- Pass from the Python scope to the Taichi scope
- Must be type-hinted
- Scalar Only
- Pass by value

```
import taichi as ti

ti.init(arch=ti.cpu)

@ti.kernel
def my_kernel(x: ti.i32, y: ti.f32):
    print(x + y)

my_kernel(2, 3.3)  # 5.3
```

Kernel arguments

- At most 8 parameters
- Pass from the Python scope to the Taichi scope
- Must be type-hinted
- Scalar Only
- Pass by value

```
import taichi as ti

ti.init(arch=ti.cpu)

@ti.kernel
def bad_kernel(v: ti.Vector):
    ...

@ti.kernel
def good_kernel(vx: ti.f32, vy: ti.f32):
    v = ti.Vector([vx, vy])
    ...
```

Kernel arguments

- At most 8 parameters
- Pass from the Python scope to the Taichi scope
- Must be type-hinted
- Scalar Only
- Pass by value

```
import taichi as ti

ti.init(arch=ti.cpu)

@ti.kernel
def foo(x: ti.i32):
    x = x + 1
    print("x in foo:", x) # 101

x = 100
foo(x)
print("x outside foo:", x) # 100
```

Return value of a @ti.kernel

- May or may not return
- Returns one single scalar value only
- Must be type-hinted

```
import taichi as ti

ti.init(arch=ti.cpu)

@ti.kernel
def my_kernel() -> ti.i32: # returns int32
    return 233.666

print(my_kernel()) # ?
```

Functions

- A Python function decorated by @ti.func is a Taichi function
  - Taichi functions can only be called **from the Taichi scope**

```
import taichi as ti

ti.init(arch=ti.cpu)

def foo():
    print("foo")
    bar()

@ti.func
def bar():
    print("bar")

foo()
```

```
import taichi as ti

ti.init(arch=ti.cpu)

@ti.kernel
def foo():
    print("foo")
    bar()

@ti.func
def bar():
    print("bar")

foo()
```

Functions

- Taichi functions can be nested
- Taichi functions are force-inlined

```
import taichi as ti

ti.init(arch=ti.cpu)

@ti.kernel
def foo():
    print("foo")
    bar()

@ti.func
def bar():
    print("bar")
    baz()

@ti.func
def baz():
    print("baz")

foo()
```

Functions

- Taichi functions can be nested
- Taichi functions are **force-inlined**

```
import taichi as ti

ti.init(arch=ti.cpu)

@ti.kernel
def foo():
    print("foo")
    bar(1)

@ti.func
def bar(i):
    print("bar")
    if i != 0:
        bar(0)

foo()
```

Arguments and return values

- Do not need to be type-hinted
- Arguments pass by value

Arguments and return values

- Do not need to be type-hinted
- Arguments pass by value

```python
import taichi as ti

ti.init(arch=ti.cpu)

@ti.func
def my_add(x, y):
    return x + y

@ti.kernel
def my_kernel():
    ret = my_add(2, 3.3)
    print(ret)  # 5.3

my_kernel()
```

```python
import taichi as ti

ti.init(arch=ti.cpu)

@ti.func
def my_add(vec):
    return vec[0]+vec[1]

@ti.kernel
def my_kernel():
    ret = my_add(ti.Vector([2, 3.3]))
    print(ret)  # 5.3

my_kernel()
```

```python
import taichi as ti

ti.init(arch=ti.cpu)

@ti.func
def foo(vec):
    return vec[0], vec[1]

@ti.kernel
def my_kernel():
    x, y = foo(ti.Vector([2, 3.3]))
    print(x, y)  # 2, 3.3

my_kernel()
```

Arguments and return values

- Do not need to be type-hinted
- Arguments pass by **value** (yes, for the force-inlined @ti.func)

```
import taichi as ti

ti.init(arch=ti.cpu)

@ti.func
def my_func(x):
    x = x + 1
    print("x in my func:", x)  # 234

@ti.kernel
def my_kernel():
    x = 233
    my_func(x)
    print("x outside my func:", x)  # 233

my_kernel()
```

Arguments and return values

- Do not need to be type-hinted
- Arguments pass by **value** (yes, for the force-inlined @ti.func)

```python
import taichi as ti

ti.init(arch=ti.cpu)

@ti.func
def my_func(x):
    x = x + 1
    print("x in my func:", x)  # 234
    return x

@ti.kernel
def my_kernel():
    x = 233
    y = my_func(x)
    print("x outside my func:", x)  # 233
    print("y outside my func:", y)  # 234

my_kernel()
```

Anything in @ti.kernel and @ti.func are in the **Taichi scope**

- **Static** data type in the Taichi scope
- **Static** lexical scope in the Taichi scope

Anything in @ti.kernel and @ti.func are in the **Taichi scope**

- **Static** data type in the Taichi scope
- **Static** lexical scope in the Taichi scope

```
import taichi as ti

ti.init(arch=ti.cpu)

@ti.kernel
def  err_change_types_without_implicit_cast():
    a = 1  # a = 1, initialized as an int
    a = 2.7 # a = 2, because of the implicit cast
    a = ti.Vector([1.0, 0.0])

err_change_types_without_implicit_cast()
```

Anything in @ti.kernel and @ti.func are in the **Taichi scope**

- **Static** data type in the Taichi scope
- **Static** lexical scope in the Taichi scope

```
import taichi as ti

ti.init(arch=ti.cpu)

@ti.kernel
def err_out_of_scope(x: float):
    if x < 0:
        y = -x
    else:
        y = x
    print(y)  # `y` is out of the scope

err_out_of_scope()
```

Anything in @ti.kernel and @ti.func are in the **Taichi scope**

- The @ti.kernels and @ti.func will be compiled JIT.
- The Taichi scope variables can not see the Python scope variables at run time.

```python
import taichi as ti

ti.init(arch=ti.cpu)

a = 42

@ti.kernel
def print_a():
    print('a=', a)

print_a()  # "a= 42"
a = 53
print('a=', a) # "a= 53"
print_a()  # "a= 42"
```

```python
import taichi as ti

ti.init(arch=ti.cpu)

a = ti.field(ti.i32, shape=())

@ti.kernel
def print_a():
    print('a=', a[None])

a[None] = 42
print_a()  # "a= 42"
a[None] = 53
print_a()  # "a= 53"
```

Math ops

- Taichi supports most math operations in Python

```
ti.sin(x)
ti.cos(x)
ti.asin(x)
ti.acos(x)
ti.atan2(y, x)
ti.sqrt(x)
ti.cast(x, data_type)
```

```
ti.floor(x)
ti.ceil(x)
ti.inv(x)
ti.tan(x)
ti.tanh(x)
ti.exp(x)
ti.log(x)
```

```
ti.random(data_type)
abs(x)
int(x)
float(x)
max(x, y, ...)
min(x, y, ...)
x ** y
```

```
A.transpose()
A.inverse()
A.trace()
A.determinant()
v.normalized()

A + B, A * B, A @ B, ...
```

```
R, S = ti.polar_decompose(A, ti.f32)
U, sigma , V = ti.svd(A, ti.f32)
lam, V = ti.eig(A, ti.f32)

u.dot(v)            # scalar
u.cross(v)          # vector
u.outer_product(v)  # matrix
```

Example: N-body

```
@ti.kernel
def initialize():
    center = ti.Vector([0.5, 0.5])
    for i in range(N):
        theta = ti.random() * 2 * PI
        r = (ti.sqrt(ti.random()) * 0.7 + 0.3
) * galaxy_size
        offset = r * ti.Vector([ti.cos(theta)
, ti.sin(theta)])
        pos[i] = center+offset
        vel[i] = [-offset.y, offset.x]
        vel[i] *= init_vel
```

```
@ti.kernel
def update():
    dt = h/substepping
    for i in range(N):
        #symplectic euler
        vel[i] += dt*force[i]/m
        pos[i] += dt*vel[i]
```

```
@ti.kernel
def compute_force():
    # clear force
    for i in range(N):
        force[i] = ti.Vector([0.0, 0.0])

    # compute gravitational force
    for i in range(N):
        p = pos[i]
        for j in range(N):
            if i != j: # double the computation for a better
 memory footprint and load balance
                diff = p-pos[j]
                r = diff.norm(1e-5)

                # gravitational force -(GMm / r^2) * (diff/r
) for i
                f = -G * m * m * (1.0/r)**3 * diff

                # assign to each particle
                force[i] += f
```

Example: N-body (cont'd)

```
initialize()
while gui.running:

    for i in range(substepping):
        compute_force()
        update()

    ...
```

Visualization

Print

- “almost” your Python print

```
print(arg1, ..., sep=' ', end='\n')
```

- Print arguments support string, scalar, vector, and matrix

```
m = ti.Matrix([[2, 3, 4], [5, 6, 7]])
print('m =', m)  #=> m = [[2, 3, 4], [5, 6, 7]]
```

- Works everywhere (almost) (with terms and conditions)

Print in a @ti.kernel for the GPU backend

- Random order

```
import taichi as ti
ti.init(arch=ti.cuda)

@ti.kernel
def kern():
    for i in range(10):
        print(i)

kern()
```

Print in a @ti.kernel for the GPU backend

- The print() in GPU is not likely to show until seeing ti.sync()

```
import taichi as ti
ti.init(arch=ti.cuda)

@ti.kernel
def kern():
    print('inside kernel')

print('before kernel')
kern()
print('after kernel')
ti.sync()
print('after sync')
```

GUI

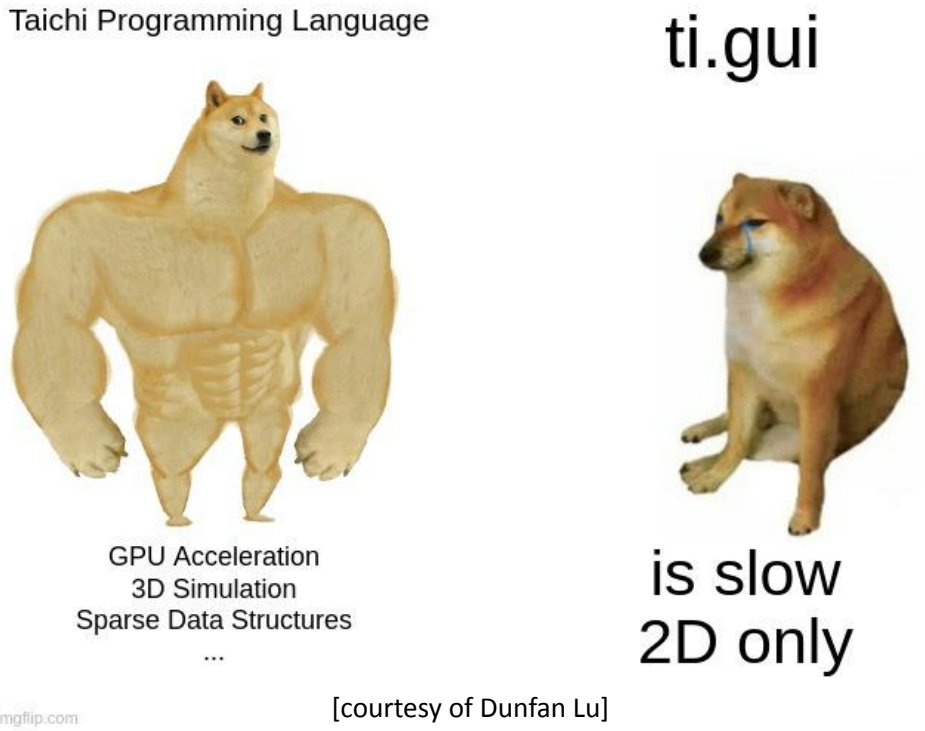
- 2D : ti.GUI
  - set your window: gui = ti.GUI(“Title”, res=(1024, 768))
  - paint on a window: gui.set_image()
  - elements : gui.circles(), gui.lines(), gui.rects(), gui.triangles(), ...
  - widgets : gui.button(), gui.slider(), gui.text(), ...
  - events : gui.get_events(), gui.get_key_event(), gui.running, ...

```
for i in range(1000000):
    paint(i * 0.01)
    gui.set_image(pixels)
    gui.show()
```

```
gui = ti.GUI('N-body problem', (512, 512))

while gui.running:

    ...

    gui.clear(0x112F41)
    gui.circles(pos.to_numpy(), color=0xffffff, radius=planet_radius)
    gui.show()
```

GUI

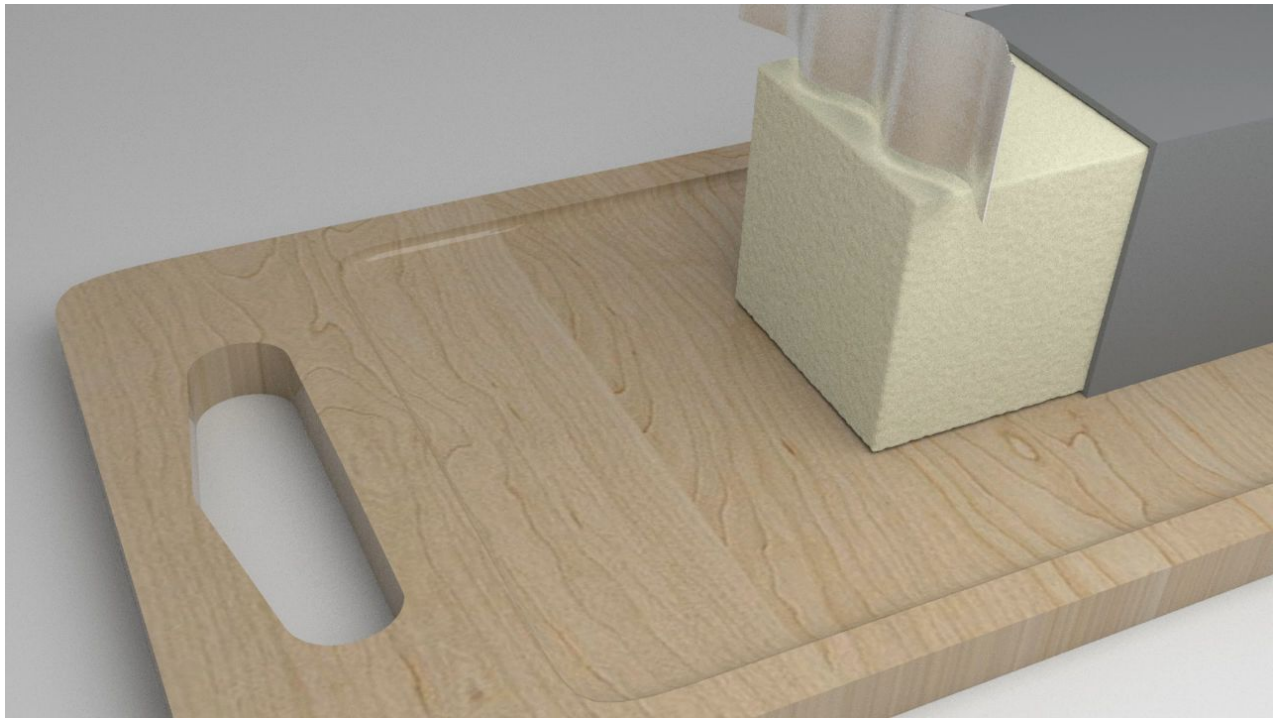
- ti.GUI: for prototyping only



[courtesy of Dunfan Lu]

GUI

- 3D
  - offline : ti.PLYWriter
  - Realtime (GPU backend only, WIP) : GGUI



Build your first Taichi Project

Data

Computation

Visualization

Init

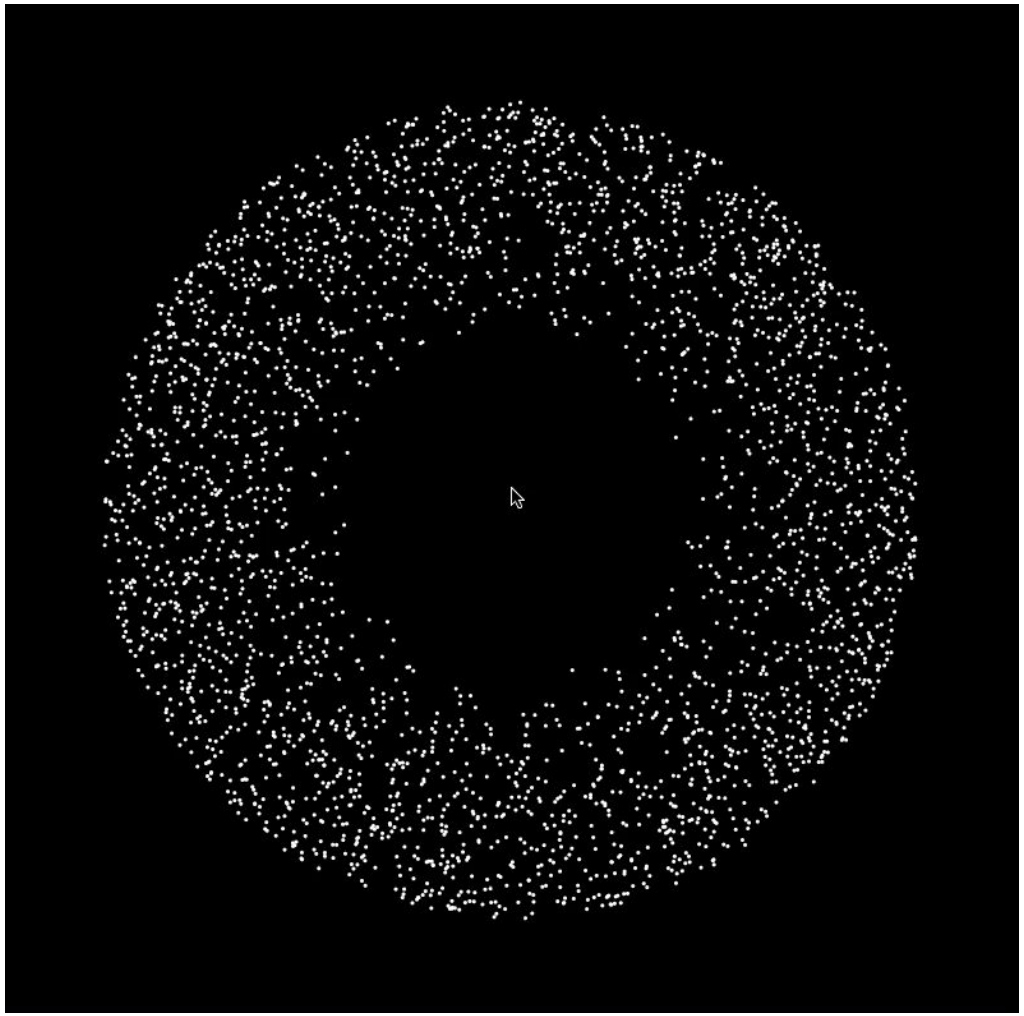
Data

Computation

Visualization

```
import taichi as ti

ti.init(ti.gpu)

# gravitational constant 6.67408e-11, using 1 for simplicity
G = 1
PI = 3.141592653

# number of planets
N = 300
# unit mass
m = 5
# galaxy size
galaxy_size = 0.4
# planet radius (for rendering)
planet_radius = 2
# init vel
init_vel = 120

# time-step size
h = 1e-5
# substepping
substepping = 10

# pos, vel and force of the planets
# Nx2 vectors
pos = ti.Vector.field(2, ti.f32, N)
vel = ti.Vector.field(2, ti.f32, N)
force = ti.Vector.field(2, ti.f32, N)

@ti.kernel
def initialize():
    center = ti.Vector([0.5, 0.5])
    for i in range(N):
        theta = ti.random() * 4 * PI
        r = (ti.sqrt(ti.random()) * 0.7 + 0.3) * galaxy_size
        offset = r * ti.Vector([ti.cos(theta), ti.sin(theta)])
        pos[i] = center+offset
        vel[i] = [-offset.y, offset.x]
        vel[i] *= init_vel

@ti.kernel
def compute_force():
    # clear force
    for i in range(N):
        force[i] = ti.Vector([0.0, 0.0])

    # compute gravitational force
    for i in range(N):
        p = pos[i]
        for j in range(N):
            if i != j: # double the computation for a better memory footprint and load balance
                diff = p-pos[j]
                r = diff.norm(1e-5)

                # gravitational force -(GMm / r^2) * (diff/r) for i
                f = -G * m * m * (1.0/r)**3 * diff

                # assign to each particle
                force[i] += f

@ti.kernel
def update():
    dt = h/substepping
    for i in range(N):
        #symplectic euler
        vel[i] += dt*force[i]/m
        pos[i] += dt*vel[i]

gui = ti.GUI('N-body problem', (512, 512))

initialize()
while gui.running:

    for i in range(substepping):
        compute_force()
        update()

    ...

    gui.clear(0x112F41)
    gui.circles(pos.to_numpy(), color=0xffffff, radius=planet_radius)
    gui.show()
```

Hello World @ Taichi



```python
import taichi as ti

ti.init(ti.gpu)

# gravitational constant 6.67408e-11, using 1 for simplicity
G = 1
PI = 3.141592653

# number of planets
N = 300
# unit mass
m = 5
# galaxy size
galaxy_size = 0.4
# planet radius (for rendering)
planet_radius = 2
# init vel
init_vel = 120

# time-step size
h = 1e-5
# substepping
substepping = 10

# pos, vel and force of the planets
# Nx2 vectors
pos = ti.Vector.field(2, ti.f32, N)
vel = ti.Vector.field(2, ti.f32, N)
force = ti.Vector.field(2, ti.f32, N)

@ti.kernel
def initialize():
    center = ti.Vector([0.5, 0.5])
    for i in range(N):
        theta = ti.random() * 2 * PI
        r = (ti.sqrt(ti.random()) * 0.7 + 0.3) * galaxy_size
        offset = r * ti.Vector([ti.cos(theta), ti.sin(theta)])
        pos[i] = center+offset
        vel[i] = [-offset.y, offset.x]
        vel[i] *= init_vel

@ti.kernel
def compute_force():
    # clear force
    for i in range(N):
        force[i] = ti.Vector([0.0, 0.0])

    # compute gravitational force
    for i in range(N):
        p = pos[i]
        for j in range(N):
            if i != j: # double the computation for a better memory footprint and load balance
                diff = p-pos[j]
                r = diff.norm(1e-5)

                # gravitational force -(GMm / r^2) * (diff/r) for i
                f = -G * m * m * (1.0/r)**3 * diff

                # assign to each particle
                force[i] += f

@ti.kernel
def update():
    dt = h/substepping
    for i in range(N):
        #symplectic euler
        vel[i] += dt*force[i]/m
        pos[i] += dt*vel[i]

gui = ti.GUI('N-body problem', (512, 512))

initialize()
while gui.running:

    for i in range(substepping):
        compute_force()
        update()

    ...

    gui.clear(0x112F41)
    gui.circles(pos.to_numpy(), color=0xffffff, radius=planet_radius)
    gui.show()
```

The N-body problem: https://zoo.taichi.graphics/playground/e9baa99382789771692 80ba40355d302

Remark

- Taichi scope v.s. Python scope
  - Taichi scope is “on device”, “statically-typed”, “strongly-typed”

Remark

- Taichi scope v.s. Python scope
  - Taichi scope is “on device”, “statically-typed”, “strongly-typed”
- @ti.kernel
  - “__global__” functions in CUDA
  - The **outermost** scope for-loop in a @ti.kernel is parallelized
- @ti.func
  - “__device__” functions in CUDA
  - force-inlined

Remark

- Taichi scope v.s. Python scope
  - Taichi scope is “on device”, “statically-typed”, “strongly-typed”
- @ti.kernel
  - “__global__” functions in CUDA
  - The **outermost** scope for-loop in a @ti.kernel is parallelized
- @ti.func
  - “__device__” functions in CUDA
  - force-inlined
- Pass-by-value

Remark

- Taichi scope v.s. Python scope
  - Taichi scope is “on device”, “statically-typed”, “strongly-typed”
- @ti.kernel
  - “__global__” functions in CUDA
  - The **outermost** scope for-loop in a @ti.kernel is parallelized
- @ti.func
  - “__device__” functions in CUDA
  - force-inlined
- Pass-by-value
- Docs: https://docs.taichi.graphics/docs/

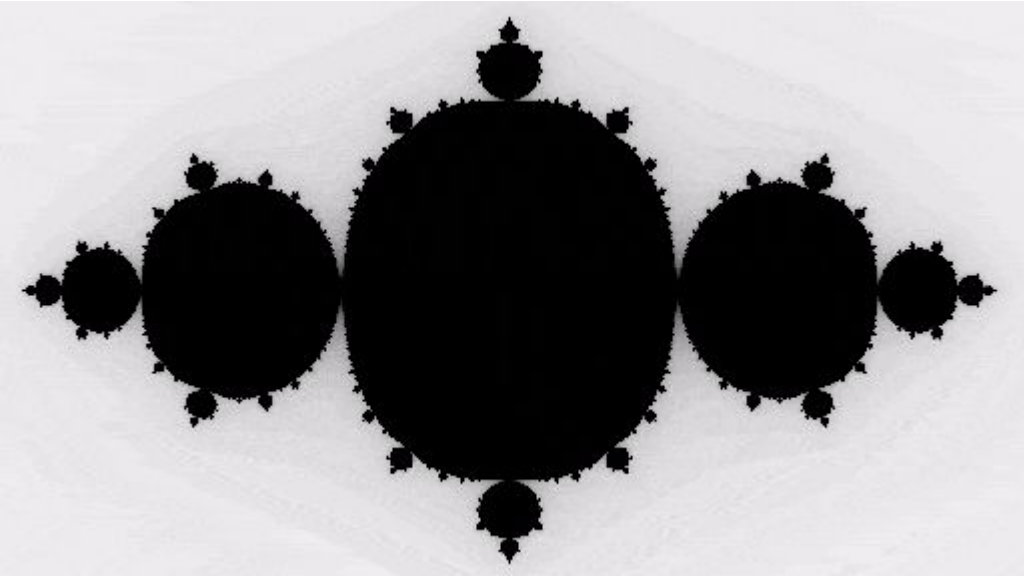
Share your awesome work (optional)

- Could be ANYTHING you programmed using Taichi

- Share it with your classmates at forum.taichi.graphics
  - 太极图形课作业区: https://forum.taichi.graphics/c/homework/14
  - Share your Taichi zoo link or your github/gitee link
  - Compile a .gif animation at your will

Julia-set

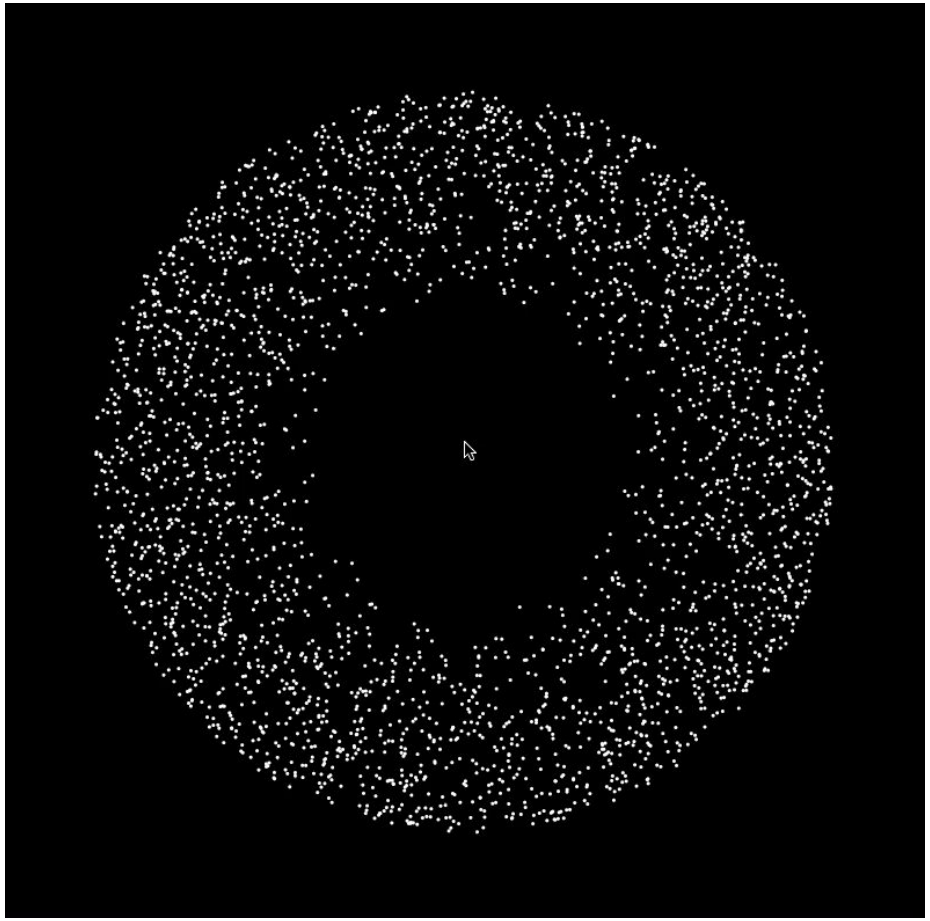
https://zoo.taichi.graphics/playground/e9baa99382789771692 80ba40355d302



- Understand the **workflow** of this fractal animation
- Change the **shape** of this fractal animation
- Change the **color** of this animation to RGB

N-body problem

https://zoo.taichi.graphics/playground/e9baa99382789771692 80ba40355d302



- **GUI:** Add a mouse event
  - E.g.: click to mimic a black hole (try your first 0-D ti.field)
- Try a local build
  - Test the difference between **ti.cpu** and **ti.gpu**

Questions?

本次答疑：09/23

下次直播：09/28

直播回放：Bilibili 搜索「太极图形」

主页&课件：https://github.com/taichiCourse01