Data-driven Computer Animation

Tutorial 5 – Introduction to Taichi

Prof. Taku Komura
TAs: Zhouyingcheng Liao(zycliao@cs.hku.hk)

What is Taichi

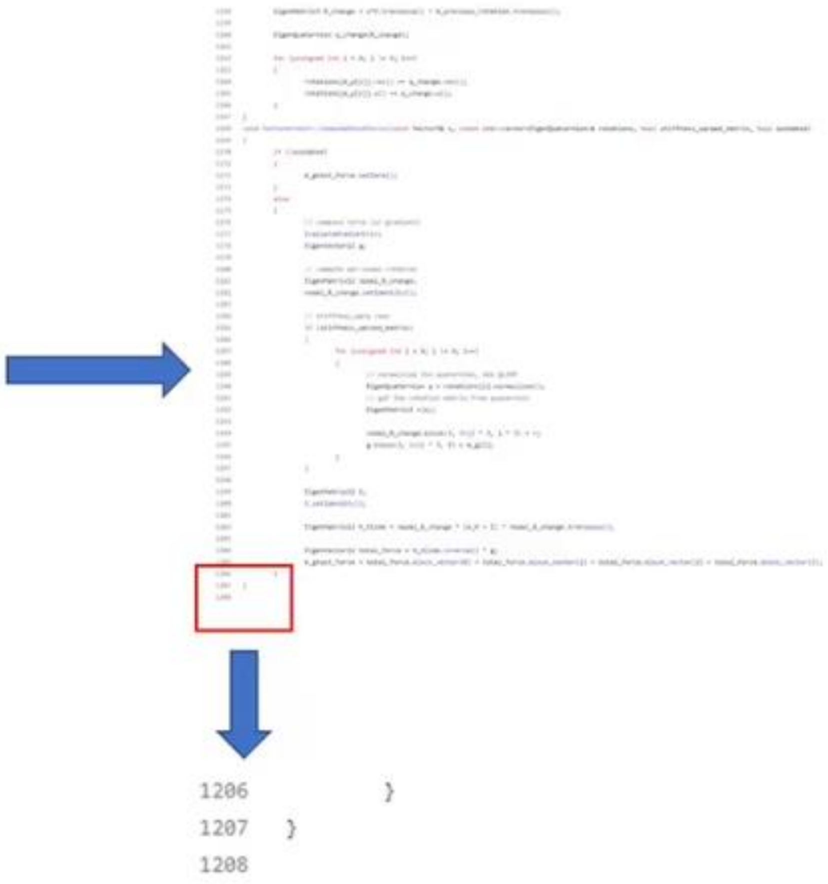
A programming language designed for computer graphics

- Productivity
  - Friendly learning curve
  - Shorter code, higher perf.
- Portability
  - Multi-backend support
- Performance
  - Optimized for bandwidth, locality and load balancing

Why Taichi

- **Productivity**
- Portability
- Performance

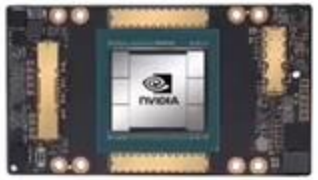
- constraint.cpp
- constraint.h
- constraint_attachment.cpp
- constraint_penalty.cpp
- constraint_spring.cpp
- constraint_tet.cpp



Why Taichi

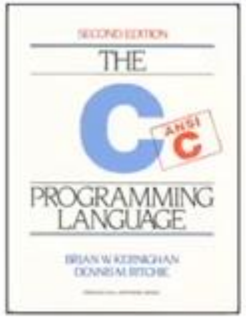
- Productivity
- Portability
- Performance





Why Taichi

- Productivity
- Portability
- Performance



ti.init()

- Entrance to all Taichi projects

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

ti.init()

- arch:
  - ti.cpu/ti.gpu/ti.arm/ti.cuda...

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

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

ti.init()

- To specify the GPU ID
  - for CUDA: export CUDA_VISIBLE_DEVICES=[gpuid]
  - for Vulkan: export TI_VISIBLE_DEVICES=[gpuid]

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

Python vs 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

Only codes in ti.kernel and ti.func are in Taichi-scope

Tachi – Data 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

Backend	`i8`	`i16`	`i32`	`i64`	`u8`	`u16`	`u32`	`u64`	`f16`	`f32`	`f64`
CPU	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
CUDA	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
OpenGL	✗	✗	✓	○	✗	✗	✗	✗	✗	✓	✓
Metal	✓	✓	✓	✗	✓	✓	✓	✗	✗	✓	✗
Vulkan	○	○	✓	○	○	○	✓	○	✓	✓	○

> ○: Requiring extensions for the backend.

Data Type Aliases

- Default types can be changed using the configuration option default_ip and default_fp

```
ti.init(default_ip=ti.i64, default_fp=ti.f64)

@ti.kernel
def example_cast() -> int:  # the returned type is ti.i64
    x = 3.14    # x is of ti.f64 type
    y = int(x)  # equivalent to ti.i64(x)
    return y
```

Data Type Aliases

- Do not mix up Taichi’s int and other int

```
x = numpy.array([1, 2, 3, 4], dtype=int)  # NumPy's int64 type
y = int(3.14)  # Python's built-in int type
```

Type Casts

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

```python
import taichi as ti

ti.init(arch=ti.cpu)

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

foo() #2.7
```

```python
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 - Indexing

- 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

- “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=())
```

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

```python
import taichi as ti

ti.init(arch=ti.cpu)

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

pixels[1, 2] = 42.0
```

```python
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

- "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.grouped()

- Taichi provides ti.grouped syntax which supports grouping loop indices into a ti.Vector.
- It enables dimensionality-independent programming, i.e., code are adaptive to scenarios of different dimensionalities automatically

```
# without ti.grouped
for I in ti.ndrange(2, 3):
    print(I)
prints 0, 1, 2, 3, 4, 5
```

```
# with ti.grouped
for I in ti.grouped(ndrange(2, 3)):
    print(I)
prints [0, 0], [0, 1], [0, 2], [1, 0], [1, 1], [1, 2]
```

ti.grouped()

- Taichi provides ti.grouped syntax which supports grouping loop indices into a ti.Vector.
- It enables dimensionality-independent programming, i.e., code are adaptive to scenarios of different dimensionalities automatically

```python
import taichi as ti
ti.init()

a = ti.Matrix.field(n=2, m=3, dtype=ti.f32, shape=(2, 2))
@ti.kernel
def test():
    for i in ti.grouped(a):
        # a[i] is a 2x3 matrix
        a[i] = [[1, 1, 1], [1, 1, 1]]
```

Matrix Size Consideration

- Matrix operations are unrolled at compile time. For performance reasons, it is recommended that you keep your matrices small.

```python
import taichi as ti
ti.init()

a = ti.Matrix.field(n=2, m=3, dtype=ti.f32, shape=(2, 2))
@ti.kernel
def test():
    for i in ti.grouped(a):
        # a[i] is a 2x3 matrix
        a[i] = [[1, 1, 1], [1, 1, 1]]
        # The assignment is unrolled to the following at compile time:
        # a[i][0, 0] = 1
        # a[i][0, 1] = 1
        # a[i][0, 2] = 1
        # a[i][1, 0] = 1
        # a[i][1, 1] = 1
        # a[i][1, 2] = 1
```

Matrix Size Consideration

- Matrix operations are unrolled at compile time. For performance reasons, it is recommended that you keep your matrices small.

```python
import taichi as ti
ti.init()

a = ti.Matrix.field(n=2, m=3, dtype=ti.f32, shape=(2, 2))
@ti.kernel
def test():
    for i in ti.grouped(a):
        # a[i] is a 2x3 matrix
        a[i] = [[1, 1, 1], [1, 1, 1]]
        # The assignment is unrolled to the following at compile time:
        # a[i][0, 0] = 1
        # a[i][0, 1] = 1
        # a[i][0, 2] = 1
        # a[i][1, 0] = 1
        # a[i][1, 1] = 1
        # a[i][1, 2] = 1
```

Matrix Size Consideration

- Workaround: When declaring a matrix field, leave large dimensions to the fields, rather than to the matrices. If you have a 3x2 field of 64x32 matrices:

- Not recommended: `ti.Matrix.field(64, 32, dtype=ti.f32, shape=(3, 2))`
- Recommended: `ti.Matrix.field(3, 2, dtype=ti.f32, shape=(64, 32))`

Computation Kernel

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

```python
import taichi as ti

ti.init(arch=ti.cpu)

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

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

foo()
bar()
```

```python
import taichi as ti

ti.init(arch=ti.cpu)

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

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

foo()
```

```python
import taichi as ti

ti.init(arch=ti.cpu)

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

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

foo()
```

Loops

- 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
```

Loops

- Outermost scope ?

```python
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 :-(
            ...
```

Loops

- Design your for loops for best performance

```python
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()
```

```python
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()
```

Loops

- 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!
```

Loops

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

```python
@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]
```

Loops

- 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

```python
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()
```

```python
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()
```

Loops

- 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

```python
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()
```

```python
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

A kernel can accept multiple arguments.

However, it's important to note that you can't pass arbitrary Python objects to a kernel.

Python objects can be dynamic and may contain data that the Taichi compiler cannot recognize.

Kernel Arguments

- Scalars
- ti.types.matrix()
- ti.types.vector()
- ti.types.struct()
- ti.types.ndarray()
- …

Kernel Arguments

- Passed by value
  - ti.types.matrix()
  - ti.types.vector()
  - ti.types.struct()
- Passed by reference
  - ti.types.ndarray()
  - ti.template()

Kernel Arguments

- Must have type hint

```
transform_type = ti.types.struct(R=ti.math.mat3, T=ti.math.vec3)
pos_type = ti.types.struct(x=ti.math.vec3, trans=transform_type)

@ti.kernel
def kernel_with_nested_struct_arg(p: pos_type) -> ti.math.vec3:
    return p.trans.R @ p.x + p.trans.T

trans = transform_type(ti.math.mat3(1), [1, 1, 1])
p = pos_type(x=[1, 1, 1], trans=trans)
print(kernel_with_nested_struct_arg(p))  # [4., 4., 4.]
```

Kernel Return Value

- Must have the type hint
- Could either be a scalar, ti.types.matrix(), or ti.types.vector()
- In CPU and CUDA backend, could also be ti.types.struct()
- If the return value is a vector or matrix, please ensure that it contains no more than 32 elements. (Warning otherwise)

Kernel Return Value

- At most one return value

```python
vec2 = ti.math.vec2

@ti.kernel
def test(x: float, y: float) -> vec2: # Return value must be type hinted
    # Return x, y  # Compilation error: Only one return value is allowed
    return vec2(x, y)  # Fine
```

- At most one return statement

```python
@ti.kernel
def test_sign(x: float) -> float:
    if x >= 0:
        return 1.0
    else:
        return -1.0
    # Error: multiple return statements
```

Taichi Function

- Taichi functions are fundamental units of a kernel and can only be called from within a kernel or another Taichi function.

Kernel vs Function

	Kernel	Taichi Function
Call scope	Python scope	Taichi scope
Type hint arguments	Mandatory	Recommended
Type hint return values	Mandatory	Recommended
Return type	• Scalar • `ti.types.matrix()` • `ti.types.vector()` • `ti.types.struct()`(Only on LLVM-based backends)	• Scalar • `ti.types.matrix()` • `ti.types.vector()` • `ti.types.struct()` • ...
Maximum number of elements in arguments	• 32 (OpenGL) • 64 (otherwise)	Unlimited
Maximum number of return values in a return statement	1	Unlimited