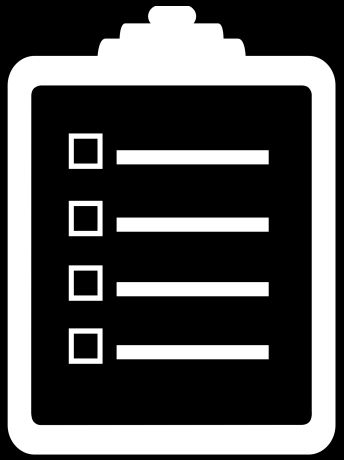
Particle Simulator

Ali Malik
malikali@stanford.edu

Game Plan



Recap

Time-driven Simulation

Event-driven Simulation

ParticleSimulator

Recap

Operator Overloading

Let’s go back for a second…

+	-	*	/	%	^
&	|	~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	||
+=	-=	*=	/=	%=	^=
&=	|=	<<=	>>=	[]	()
->	->*	new	new []	delete	delete []

Operator Overloading

Two ways to overload operators:

- Member functions
- Non-member functions

Member Functions

Just add a function named operator@ to your class

```
bool operator==(const HashSet& rhs) const;
Set operator+(const Set& rhs) const;
Set& operator+=(const ValueType& value);
```

For binary operators, accept the right hand side as an argument.

I usually name mine rhs.

Non-member Functions

Add a function named operator@ outside your class.

Have it take all its operands.

```
bool operator==(const Point& lhs, const Point& rhs) {
    return lhs.x == rhs.x && lhs.y == rhs.y;
}
```

Operator Overloading

Let’s go back for a second…

Anything curious here?

+	-	*	/	%	^
&	|	~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	||
+=	-=	*=	/=	%=	^=
&=	|=	<<=	>>=	[]	()
->	->*	new	new []	delete	delete []

Operator Overloading

Let’s go back for a second…

Anything curious here?

+	-	*	/	%	^
&	|	~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	||
+=	-=	*=	/=	%=	^=
&=	|=	<<=	>>=	[]	()
->	->*	new	new []	delete	delete []

Operator Overloading

Let’s go back for a second…

Anything curious here?

+	-	*	/	%	^
&	|	~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	||
+=	-=	*=	/=	%=	^=
&=	|=	<<=	>>=	[]	()
->	->*	new	new []	delete	delete []

Functors

Classes which define the `()` operator.

Why is this useful?

- Can have state
- Customizable through constructor

Very useful for algorithms!

Functors

Functors let us make customizable functions!

We can pass useful information to their constructor that was not known at compile time.

But...

Kind of a Pain™

C++ has a solution!

Functors

Functors let us make customizable functions!

We can pass useful information to their constructor that was not known at compile time.

But...

Kind of a Pain™

C++11 has a solution!

Lambdas

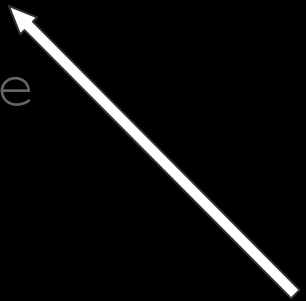
A C++11 feature that lets you make functions on the fly.

```
[capture-list](params) -> ReturnType {
    // code
};
```

Lambda Captures

A C++11 feature that lets you make functions on the fly.

```
[capture-list](params) -> ReturnType {
    // code
};
```



What is this for?

Lambda Captures

You can capture available variables to use in the lambda

```
[byValue, &byReference]
```

You can also capture all currently available variables:

```
[=] // By value

[&] // By reference
```

This will only capture the ones used inside the function.

How Lambdas Work?

```
[capture-list](params) ->
ReturnType {

    // code

};
```

```
class SomeName {
public:
    SomeName(capture-list) {
        // set each private member to
        // thing in capture list
    }

    ReturnType operator()(params) {
        // code
    }

private:
    // create private member for each
    // thing in capture-list
};
```

Particle Simulator

Particle Simulator

Useful for understanding physical systems (diffusion, reactions etc.)

Model:

- n spherical particles in a box
- Each particle has position (x, y) and velocity (vx, vy)
- Collisions are elastic
- No other forces in the system

Particle Class

How do we design the particle class?

Particle Class

Particle Class

Internal members:

- position (`double x, y`)
- velocity (`double vx, vy`)
- radius (`double radius`)
- mass (`double mass`)

Accessor methods

`getX(), getY(), getVx(), getVy(), getRadius(), getMass()`

Useful modifying methods

```
void move(double dt)
void bounceOff(Particle* other)
void bounceOffVerticalWall()
void bounceOffHorizontalWall()
```

Useful querying methods

```
double timeToHit(Particle* other)
double timeToHitVerticalWall()
double timeToHitHorizontalWall()
```

Particle Class

One caveat: we will use pointers

Implement Particle Class
(ParticleSimulator.pro)

Particle Simulator

How do we implement collisions?

```
while true:
    Move each particle by a small dt
    For any pair of particles that have collided, change their vx, vy
    For any particles colliding with walls, change their vx, vy
    Draw canvas
    pause
```

Particle Simulator

How do we implement collisions?

```
while true:
    Move each particle by a small dt
    For any pair of particles that have collided, change their vx, vy
    For any particles colliding with walls, change their vx, vy
    Draw canvas
    pause
```

Callled a Time-driven Simulation

Particle Simulator

One caveat: we will use pointers

Simple Particle Simulator
(ParticleSimulatorSimple.pro)

Particle Simulator

Issues with Time-driven Simulations:

- Really slow! O(n^2) checks every time increment.
- Might miss collisions if dt is too large
- Smaller dt means slower code

Most of the time, we don’t have a collision but are still doing the O(n^2) checks!

Particle Simulator

Issues with Time-driven Simulations:

- Really slow! O(n^2) checks every time increment.
- Might miss collisions if dt is too large
- Smaller dt means slower code

Can we do better?

Most of the time, we don’t have a collision but are still doing the O(n^2) checks!

Event-driven Simulation

Between collisions, particles move in a constant, straight line

Focus on times when interesting events i.e. collisions happen

Pseudocode:

```
Maintain sequence of all future collisions ordered by time.
Advance time to moment of earliest collision.
Change particle involved in collision.
Invalidate any other future collisions these particles are involved in
Make predictions of new collisions
```

Event-driven Simulation

More detailed pseudocode:

Maintain event sequence of all future collisions ordered by time.

Remove earliest event from event sequence

If event was invalidated, discard it. An event is invalid if one of the particles involved in the event have collided since the event was enqueued.

Otherwise, event is a collision so:

Advance time to the moment of this collision.

Update velocities of particles involved in collision.

Make predictions of new collisions involving these particles and add to event sequence.

Event-driven Simulation

More detailed pseudocode:

```
Maintain event sequence of all future collisions ordered by time.

Remove earliest event from event sequence

If event was invalidated, discard it. An event is invalid if one of the
particles involved in the event have collided since the event was enqueued.

Otherwise, event is a collision so:

Advance time to the moment of this collision.

Update velocities of particles involved in collision.

Make predictions of new collisions involving these particles and add to event
sequence.
```

Event-driven Simulation

What collection could we use here?

More detailed pseudocode:

```
Maintain event sequence of all future collisions ordered by time.

Remove earliest event from event sequence

If event was invalidated, discard it. An event is invalid if one of the
particles involved in the event have collided since the event was enqueued.

Otherwise, event is a collision so:

Advance time to the moment of this collision.

Update velocities of particles involved in collision.

Make predictions of new collisions involving these particles and add to event
sequence.
```

std::priority_queue

A priority queue lets us get the earliest event in the sequence!

```
template<
    class T,
    class Container = std::vector<T>,
    class Compare = std::less<typename Container::value_type>
> class priority_queue;
```

std::priority_queue

Needs three template types to be constructed:

Template parameters

T - The type of the stored elements. The behavior is undefined if T is not the same type as Container::value_type. (since C++17)

Container - The type of the underlying container to use to store the elements. The container must satisfy the requirements of SequenceContainer, and its iterators must satisfy the requirements of RandomAccessIterator. Additionally, it must provide the following functions with the usual semantics:

- front()
- push_back()
- pop_back()

The standard containers std::vector and std::deque satisfy these requirements.

Compare - A Compare type providing a strict weak ordering

Event-driven Simulation

More detailed pseudocode:

Maintain event sequence of all future collisions ordered by time.

Remove earliest event from event sequence

If event was invalidated, discard it. An event is invalid if one of the particles involved in the event have collided since the event was enqueued.

Otherwise, event is a collision so:

Advance time to the moment of this collision.

Update velocities of particles involved in collision.

Make predictions of new collisions involving these particles and add to event sequence.

Event Class

Internal members:

- Event time (`double eventTime`)
- First particle/null (`Particle* a`)
- Second particle/null (`Particle* b`)
- Initial collision count of a (`int countA`)
- Initial collision count of b (`int countB`)

Accessor methods

```
Particle* getFirstParticle()
Particle* getSecondParticle() double
getEventTime()
```

Useful querying methods

```
// checks if collision count of the particle is same
// as its initial collision count
bool isValid()
```

Particle Simulator

One caveat: we will use pointers

Particle Simulator
(ParticleSimulator.pro)