Week 6

Input Handling

Hooman Salamat

2019-02-10

Objectives

- Examine SFML events and explore their purpose as input
- Assess real-time input and evaluate its difference from events
- Analyze and reproduce a command-based communication system to deliver events
- Explore how to dynamically bind keys at runtime

Polling Events

- Events are objects that are triggered when something happens
  - E.g., user input
- Behind the scenes, the OS reports an event to the application
  - SFML processes such a report
  - Converts it into a corresponding SFML event type

Polling Events (cont'd.)

- Specifically, we extract events using the `sf::Window::pollEvent()` function
- It's signature is:

```
bool sf::Window::pollEvent(sf::Event& event);
```

Polling Events (cont'd.)

- Generally we want to poll an event with an event parameter as well as a bool that will tell us to keep polling the event or not
  - If there are no more of that event type to poll

Events Thus Far

- In the examples up to now, we’ve handled events in SFML thus:

```
sf::Event event;
while (window.pollEvent(event))
{
  // Handle the event
}
```

Events

- We can group events to four different categories:
  - **window**, **joystick**, **keyboard** and **mouse**

- The next few slides outline these events

Window Events

- Window events concern windows directly
- `sf::Event::Closed`
  - Occurs when the user requests that the window be closed
  - Pressing the [X] or Alt-F4 for example
  - No data associated with this event

Window Events (cont'd.)

- `sf::Event::Resized`
  - Occurs when the window is resized
  - User drags on edges to manually resize it
  - Window must be enabled to resize
  - Data type is `sf::Event::SizeEvent` that is accessed through `event.size`

Window Events (cont'd.)

- `sf::Event::LostFocus`
- `Sf::Event::GainedFocus`
  - Window is active or inactive (clicked away from)
  - No extra data for event

Joystick Events

- Whenever a joystick or gamepad changes its state
  - Each input device has an ID number

- `sf::Event::JoystickButtonPressed`
- `sf::Event::JoystickButtonReleased`
  - Data structure associated is `sf::Event::JoystickButtonEvent` with the member `event.joystickButton`

Joystick Events (cont'd.)

- `sf::Event::JoystickMoved`
  - Triggered when analog stick or D-pad moves
  - Data is `sf::Event::JoystickMoveEvent` and accessible through member `event.joystickMove`

Joystick Events (cont'd.)

- `sf::Event::JoystickConnected`
- `Sf::Event::JoystickDisconnected`
  - Data is `sf::Event::JoystickConnectEvent` and accessible through member `event.joystickConnect`

Keyboard Events

- Generates event as the primary input device for computers

- `sf::Event::KeyPressed`
  - Data structure associated is `sf::Event::KeyEvent` with the member `event.key.code`
  - `event.key.control` are Booleans that state whether a modifier is pressed
  - Key repetition can be deactivated using `sf::Window::setKeyRepeatEnabled()`

Keyboard Events (cont'd.)

- `sf::Event::KeyReleased`
  - Counterpart to `KeyPressed`
  - Similar in function
- `sf:Event:TextEntered`
  - Designed for receiving formatted text from the user
  - Data is `sf::Event::TextEvent` and accessible through `event.text`

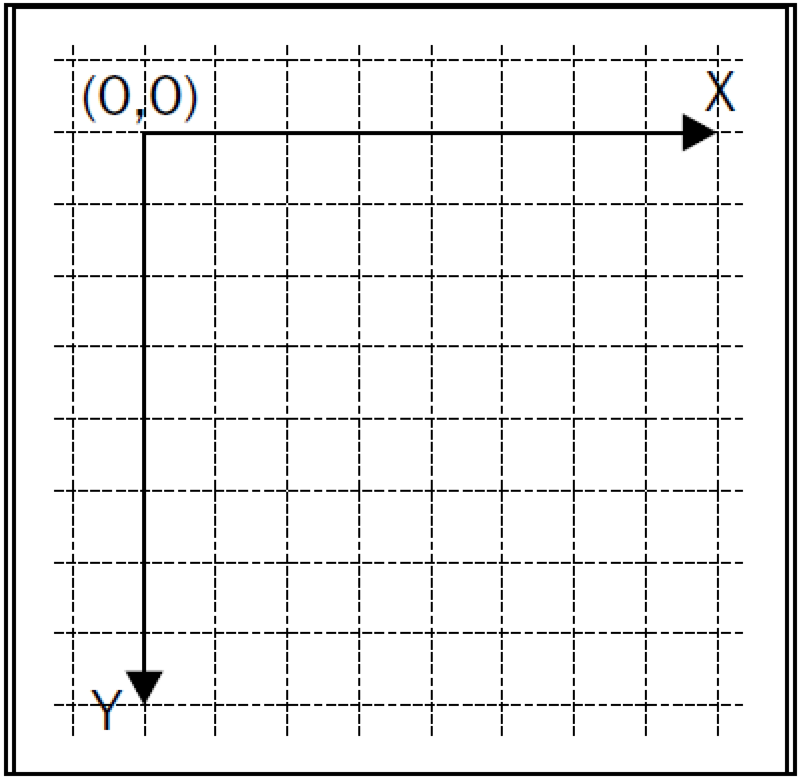
Mouse Events (cont'd.)

- Events generated when the state of the cursor, mouse buttons or mouse wheel changes

- `sf::Event::MouseEntered`
- `sf::Event::MouseLeft`
- `Sf::Event::MouseMoved`
  - Data structure for `MouseMoved` is `sf::MouseMoveEvent` and can be accessed via `event.mouseMove`

Mouse Orientation

- As most platforms, coordinates measures in window pixels



Mouse Events (cont'd.)

- `sf::Event::MouseButtonPressed`
- `sf::Event::MouseButtonReleased`
  - Data structure is `sf::MouseButtonEvent` and can be accessed via `event.mouseButton` member

- `sf::Event::MouseWheelMoved`
  - Data structure is `sf::MouseWheelEvent` and can be accessed via `event.mouseWheel` member

Handling Input

```
void Game::handlePlayerInput(sf::Keyboard::Key key, bool isPressed)
{
   if (key == sf::Keyboard::W)
      mIsMovingUp = isPressed;
   else if (key == sf::Keyboard::S)
      mIsMovingDown = isPressed;
   else if (key == sf::Keyboard::A)
      mIsMovingLeft = isPressed;
   else if (key == sf::Keyboard::D)
      mIsMovingRight = isPressed;
}
```

Handling Input (cont'd.)

```
void Game::update()
{
  sf::Vector2f movement(0.f, 0.f);
  if (mIsMovingUp)
    movement.y -= 1.f;
  if (mIsMovingDown)
    movement.y += 1.f;
  if (mIsMovingLeft)
    movement.x -= 1.f;
  if (mIsMovingRight)
    movement.x += 1.f;
  mPlayer.move(movement);
}
```

Combining Into Update

```
void Game::update(sf::Time elapsedTime)
{
  sf::Vector2f movement(0.f, 0.f);
  if (sf::Keyboard::isKeyPressed(sf::Keyboard::W))
    movement.y -= PlayerSpeed;
  if (sf::Keyboard::isKeyPressed(sf::Keyboard::S))
    movement.y += PlayerSpeed;
  if (sf::Keyboard::isKeyPressed(sf::Keyboard::A))
    movement.x -= PlayerSpeed;
  if (sf::Keyboard::isKeyPressed(sf::Keyboard::D))
    movement.x += PlayerSpeed;
  mPlayer.move(movement * elapsedTime.asSeconds());
}
```

Events vs. Real-Time Input

- If a state has changed, you should use events
- However, if you want to know the current state, then of course you must check using a function

```
// WHEN the left mouse button has been pressed, do something
if (event.type == sf::Event::MouseButtonPressed)
```

```
// WHILE the left mouse button is being pressed, do something
if (sf::Mouse::isButtonPressed(sf::Mouse::Left))
```

- So the second method is good for sustained input

Delta Movement

- The different in cursor position between two frames

```
sf::Vector2i mousePosition = sf::Mouse::getPosition(mWindow);
sf::Vector2i delta = mLastMousePosition - mousePosition;
mLastMousePosition = mousePosition;
```

Applying the Focus

```
void Game::run()
{
    while (mWindow.isOpen())
    {
       if (!mIsPaused)
         update();
       render();
       processEvents();
    }
}

void Game::processEvents()
{
    sf::Event event;
    while(mWindow.pollEvent(event))
    {
       if (event.type == sf::Event::GainedFocus)
         mIsPaused = false;
       else if (event.type == sf::Event::LostFocus)
         mIsPaused = true;
    }
}
```

Week 6

Commands

2019-02-10

Regular Functions

```
void launchMissile(int target)
{
std::cout << "Missile is launched
from regular function - target = "
<< target << '\n';
}
```

Std::function

- Class template std::function is a general-purpose polymorphic function wrapper.
- Instances of std::function can store, copy, and invoke any Callable target -- functions, lambda expressions, bind expressions, or other function objects, as well as pointers to member functions and pointers to data members.

```
std::function<void(int)> launchM1 = launchMissile;
std::cout << "store a free function:  ";
launchM1(-9);
```

Function pointers and std::function

- The difference between a regular old function pointer and std::function is that the std::function contains state.
- A function pointer is the address of an actual function defined in C++. An std::function is a wrapper that can hold any type of callable object (objects that can be used like functions).

Std::function and lambda

```
// store a lambda
std::function<void()> launchM2 =
[]() {
std::cout << "store a lambda:  " ;
launchMissile(42); };
launchM2();
```

Functor and std::function

- A functor is a class which defines the operator()
- That lets you create objects which "look like" a function
- unlike regular functions, they can contain state.

```
struct LaunchMissile {
void operator()(float target)
{
std::cout << "Missile is launched from operator - target = " << target << '\n';
}
};
```

- LaunchMissile func;
- std::function<void(float)> f(func);
- f(3);

Std::bind

- The function template bind generates a forwarding call wrapper for launchMissile .
- Calling this wrapper is equivalent to invoking launchMissile with some of its arguments bound to args.
- `std::function<void()> launchM31337 = std::bind(launchMissile, 31337);`
- `std::cout << "store the result of the call to std::bind  ";`
- `launchM31337();`

Std::bind

```
void func(int a, int b) {
// Do something important
}
```

- Consider the case when you want one of the parameters of `func` to be fixed. You can use `std::bind` to set a fixed value for a parameter
- `bind` will return a function-like object that you can place inside of `std::function`.

```
std::function<void(int)> f =
std::bind(func, _1, 5);
```

Store a call to a member function

```
struct Command {
Command(int x) : mX(x) {}
void launchMissile(float target) {
std::cout << "Missile is launched from
Command center - target = " << target <<
'\n'; }
int mX;
};
```

- `std::function<void(Command&, int)> launchM3 = &Command::launchMissile;`
- `std::cout << "store a call to member function:  ";`
- `Command command(314159);`
- `launchM3(command, 1);`

Commanding the Entities

- Some example commands might be as follows:

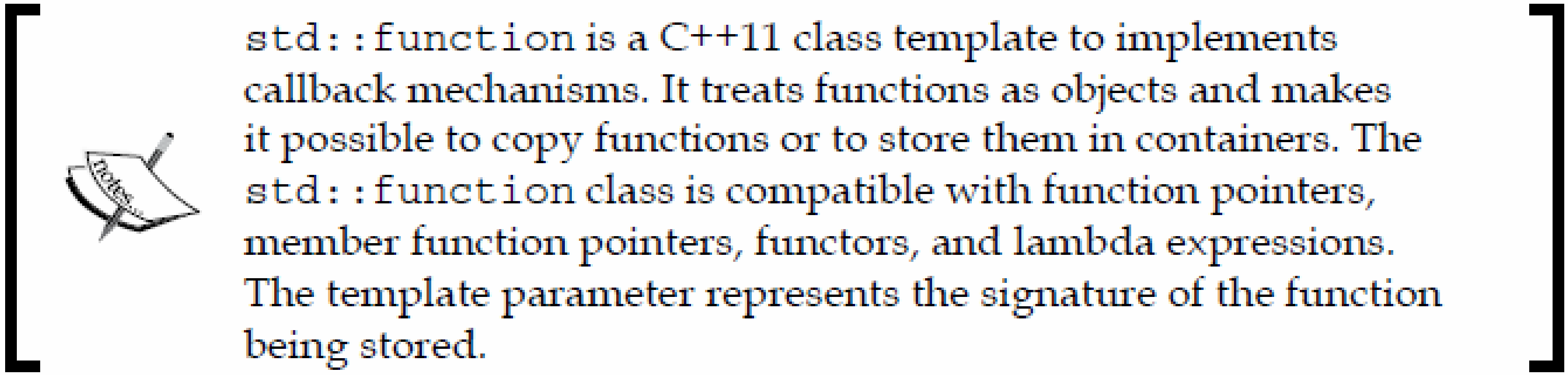
```
// One-time events
sf::Event event;
while (window.pollEvent(event))
{
   if (event.type == sf::Event::KeyPressed
   && event.key.code == sf::Keyboard::X)
     mPlayerAircraft->launchMissile();
}

// Real-time input
if (sf::Keyboard::isKeyPressed(sf::Keyboard::Left))
   mPlayerAircraft->moveLeft();
else if (sf::Keyboard::isKeyPressed(sf::Keyboard::Right))
   mPlayerAircraft->moveRight();
```

Commanding (cont'd.)

- Commands are messages that are sent to game objects
    - Alter the object
    - Issue orders:
        - Movement
        - Firing weapons
        - Triggering state changes

Command struct

```
struct Command
{
  std::function<void(SceneNode&, sf::Time)> action;
};
```



`std::function` is a C++11 class template to implements callback mechanisms. It treats functions as objects and makes it possible to copy functions or to store them in containers. The `std::function` class is compatible with function pointers, member function pointers, functors, and lambda expressions. The template parameter represents the signature of the function being stored.

std::function Example

```
int add(int a, int b) { return a + b };
std::function<int(int, int)> adder1 = &add;

std::function<int(int, int)> adder2
= [] (int a, int b) { return a + b; };
```

- Then it can be used thusly:

```
int sum = adder1(3, 5); // same as add(3, 5)
```

Movement Example

```
void moveLeft(SceneNode& node, sf::Time dt)
{
  node.move(-30.f * dt.asSeconds(), 0.f);
}
Command c;
c.action = &moveLeft;
```

- Using Lambda expression, the equivalent being:

```
c.action = [] (SceneNode& node, sf::Time dt)
{
  node.move(-30.f * dt.asSeconds(), 0.f);
};
```

- Why command over a direct function call? Because we don’t need to know on which scene node to invoke the function. We can now define any operation on a scene node.

Commanding (cont'd.)

- The different game objects should each receive their appropriate commands
- So they are divided into different categories
- Each category has one bit set to 1 and rest are set to 0

```
namespace Category
{
   enum Type
   {
      None = 0,
      Scene = 1 << 0,
      PlayerAircraft = 1 << 1,
      AlliedAircraft = 1 << 2,
      EnemyAircraft = 1 << 3,
   };
}
```

Commanding (cont'd.)

- A bitwise OR operators allows us to combine different categories, for example all airplanes:

```
unsigned int anyAircraft = Category::PlayerAircraft
                         | Category::AlliedAircraft
                         | Category::EnemyAircraft;
```

- The SceneNode class gets a new virtual method that returns the category of the game object. In the base class, we return Category::Scene by default:

```
unsigned int SceneNode::getCategory() const
{
return Category::Scene;
}
```

Commanding (cont’d.)

- `getCategory()` can be overridden to return a specific category
- an aircraft belongs to the player if it is of type Eagle, and that it is an enemy otherwise:

```
unsigned int Aircraft::getCategory() const
{
  switch (mType)
  {
      case Eagle:
            return Category::PlayerAircraft;
      default:
            return Category::EnemyAircraft;
  }
}
```

Command struct Revisited

- we give our Command class another member variable that stores the recipients of the command in a category:

```
struct Command
{
   Command();
   std::function<void(SceneNode&, sf::Time)> action;
   unsigned int category;
};
```

- The default constructor initializes the category to Category::None. By assigning a different value to it, we can specify exactly who receives the command. If we want a command to be executed for all airplanes except the player's one, the category can be set accordingly:

```
Command command;
command.action = ...;
command.category = Category::AlliedAircraft
| Category::EnemyAircraft;
```

Command Execution

- Commands are passed to the scene graph
- Inside, they are distributed to all scene nodes with the corresponding game objects
- Each scene node is responsible for forwarding a command to its children
- SceneNode::onCommand() is called everytime a command is passed to the scene graph

```
void SceneNode::onCommand(const Command& command, sf::Time dt)
{  //check if the current scene node is a receiver of the command
   if (command.category & getCategory())
      command.action(*this, dt);

   FOREACH(Ptr& child, mChildren)
      child->onCommand(command, dt);
}
```

Command Queues

- A way to transport commands to the world and the scene graph
- A class that is a very thin wrapper around a queue of commands

```
class CommandQueue
{
public:
   void push(const Command& command);
   Command pop();
   bool isEmpty() const;

private:
   std::queue<Command> mQueue;
};
```

Command Queues (cont'd.)

- The `World` class holds an instance of `CommandQueue`:

```
void World::update(sf::Time dt)
{
    ...

    // Forward commands to the scene graph
    while (!mCommandQueue.isEmpty())
      mSceneGraph.onCommand(mCommandQueue.pop(), dt);

    // Regular update step
    mSceneGraph.update(dt);
}

CommandQueue& World::getCommandQueue()
{
    return mCommandQueue;
}
```

Player and Input

- Together now we’re going to look at how the player’s input is handled
- We will look at the following:
  - The `Player` class
  - The `processInput` function from `Game`

Objectives

- Analyze and reproduce a command-based communication system to deliver events
- Explore how to dynamically bind keys at runtime

Events Thus Far

- In the examples up to now, we've handled events in SFML thus:

```
sf::Event event;
while (window.pollEvent(event))
{
  // Handle the event
}
```

Events

- We can group events to four different categories:
  - **window**, **joystick**, **keyboard** and **mouse**
- The next few slides outline these events

Window Events

- Window events concern windows directly
- `sf::Event::Closed`
  - Occurs when the user requests that the window be closed
  - Pressing the [X] or Alt-F4 for example
  - No data associated with this event

Window Events (cont'd.)

- `sf::Event::Resized`
  - Occurs when the window is resized
  - User drags on edges to manually resize it
  - Window must be enabled to resize
  - Data type is `sf::Event::SizeEvent` that is accessed through `event.size`

Window Events (cont'd.)

- `sf::Event::LostFocus`
- `Sf::Event::GainedFocus`
  - Window is active or inactive (clicked away from)
  - No extra data for event

Joystick Events

- Whenever a joystick or gamepad changes its state
  - Each input device has an ID number

- `sf::Event::JoystickButtonPressed`
- `sf::Event::JoystickButtonReleased`
  - Data structure associated is `sf::Event::JoystickButtonEvent` with the member `event.joystickButton`

Joystick Events (cont'd.)

- `sf::Event::JoystickMoved`
  - Triggered when analog stick or D-pad moves
  - Data is `sf::Event::JoystickMoveEvent` and accessible through member `event.joystickMove`

Joystick Events (cont'd.)

- `sf::Event::JoystickConnected`
- `Sf::Event::JoystickDisconnected`
  - Data is `sf::Event::JoystickConnectEvent` and accessible through member `event.joystickConnect`

Keyboard Events

- Generates event as the primary input device for computers

- `sf::Event::KeyPressed`
  - Data structure associated is `sf::Event::KeyEvent` with the member `event.key.code`
  - `event.key.control` are Booleans that state whether a modifier is pressed
  - Key repetition can be deactivated using `sf::Window::setKeyRepeatEnabled()`

Keyboard Events (cont'd.)

- `sf::Event::KeyReleased`
  - Counterpart to `KeyPressed`
  - Similar in function
- `sf:Event:TextEntered`
  - Designed for receiving formatted text from the user
  - Data is `sf::Event::TextEvent` and accessible through `event.text`

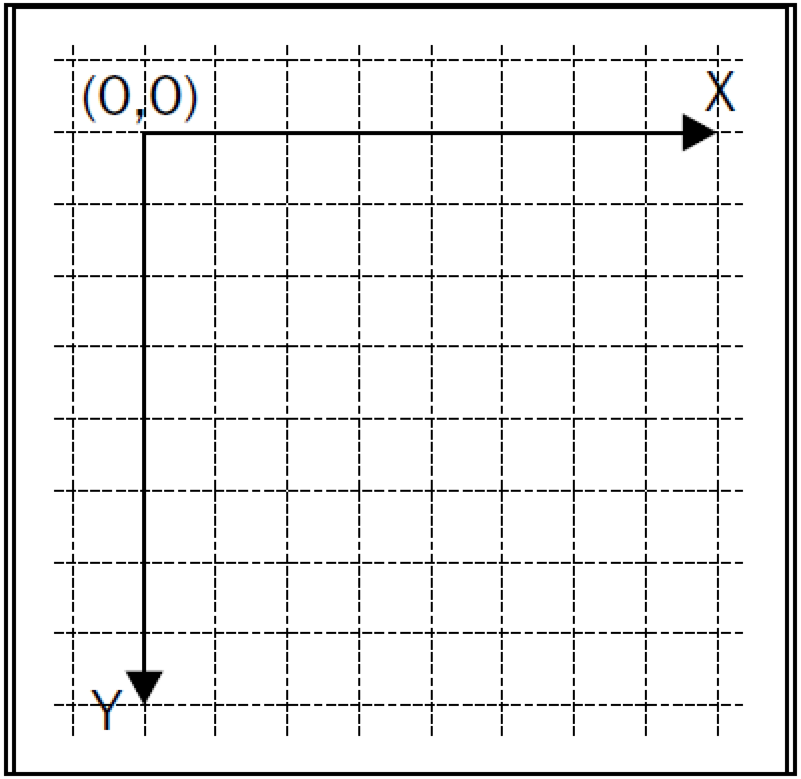
Mouse Events (cont'd.)

- Events generated when the state of the cursor, mouse buttons or mouse wheel changes

- `sf::Event::MouseEntered`
- `sf::Event::MouseLeft`
- `Sf::Event::MouseMoved`
  - Data structure for `MouseMoved` is `sf::MouseMoveEvent` and can be accessed via `event.mouseMove`

Mouse Orientation

- As most platforms, coordinates measures in window pixels



Mouse Events (cont'd.)

- `sf::Event::MouseButtonPressed`
- `sf::Event::MouseButtonReleased`
  - Data structure is `sf::MouseButtonEvent` and can be accessed via `event.mouseButton` member

- `sf::Event::MouseWheelMoved`
  - Data structure is `sf::MouseWheelEvent` and can be accessed via `event.mouseWheel` member

Handling Input

```
void Game::handlePlayerInput(sf::Keyboard::Key key, bool isPressed)
{
   if (key == sf::Keyboard::W)
     mIsMovingUp = isPressed;
   else if (key == sf::Keyboard::S)
     mIsMovingDown = isPressed;
   else if (key == sf::Keyboard::A)
     mIsMovingLeft = isPressed;
   else if (key == sf::Keyboard::D)
     mIsMovingRight = isPressed;
}
```

Delta Movement

- The different in cursor position between two frames

```
sf::Vector2i mousePosition = sf::Mouse::getPosition(mWindow);
sf::Vector2i delta = mLastMousePosition - mousePosition;
mLastMousePosition = mousePosition;
```

Applying the Focus

```
void Game::run()
{
    while (mWindow.isOpen())
    {
       if (!mIsPaused)
         update();
       render();
       processEvents();
    }
}

void Game::processEvents()
{
    sf::Event event;
    while(mWindow.pollEvent(event))
    {
       if (event.type == sf::Event::GainedFocus)
         mIsPaused = false;
       else if (event.type == sf::Event::LostFocus)
         mIsPaused = true;
    }
}
```

Commanding the Entities

- Some example commands might be as follows:

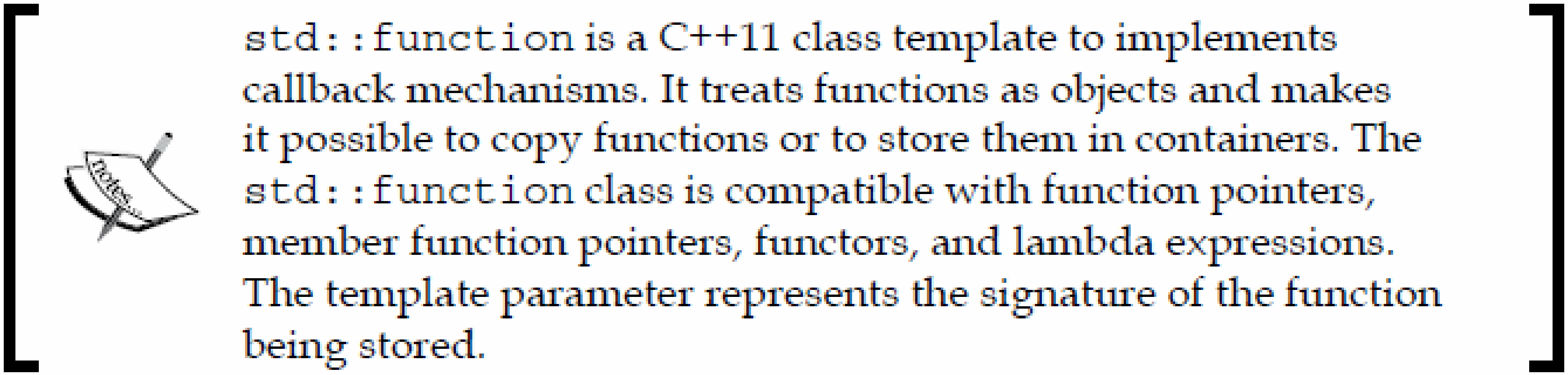
```
// One-time events
sf::Event event;
while (window.pollEvent(event))
{
   if (event.type == sf::Event::KeyPressed
   && event.key.code == sf::Keyboard::X)
     mPlayerAircraft->launchMissile();
}

// Real-time input
if (sf::Keyboard::isKeyPressed(sf::Keyboard::Left))
   mPlayerAircraft->moveLeft();
else if (sf::Keyboard::isKeyPressed(sf::Keyboard::Right))
   mPlayerAircraft->moveRight();
```

Commanding (cont'd.)

- Commands are messages that are sent to game objects
  - Alter the object
  - Issue orders:
    - Movement
    - Firing weapons
    - Triggering state changes

Command struct

```
struct Command
{
  std::function<void(SceneNode&, sf::Time)> action;
};
```



std::function is a C++11 class template to implements callback mechanisms. It treats functions as objects and makes it possible to copy functions or to store them in containers. The std::function class is compatible with function pointers, member function pointers, functors, and lambda expressions. The template parameter represents the signature of the function being stored.

std::function Example

```
int add(int a, int b) { return a + b };
std::function<int(int, int)> adder1 = &add;

std::function<int(int, int)> adder2
= [] (int a, int b) { return a + b; };
```

- Then it can be used thusly:

```
int sum = adder1(3, 5); // same as add(3, 5)
```

Movement Example

```
void moveLeft(SceneNode& node, sf::Time dt)
{
  node.move(-30.f * dt.asSeconds(), 0.f);
}
Command c;
c.action = &moveLeft;
```

- Using Lambda expression, the equivalent being:

```
c.action = [] (SceneNode& node, sf::Time dt)
{
  node.move(-30.f * dt.asSeconds(), 0.f);
};
```

- Why command over a direct function call? Because we don’t need to know on which scene node to invoke the function. We can now define any operation on a scene node.

Commanding (cont'd.)

- The different game objects should each receive their appropriate commands
- So they are divided into different categories
- Each category has one bit set to 1 and rest are set to 0

```
namespace Category
{
    enum Type
    {
        None = 0,
        Scene = 1 << 0,
        PlayerAircraft = 1 << 1,
        AlliedAircraft = 1 << 2,
        EnemyAircraft = 1 << 3,
    };
}
```

Commanding (cont'd.)

- A bitwise OR operators allows us to combine different categories, for example all airplanes:

```
unsigned int anyAircraft = Category::PlayerAircraft
                         | Category::AlliedAircraft
                         | Category::EnemyAircraft;
```

- The SceneNode class gets a new virtual method that returns the category of the game object. In the base class, we return Category::Scene by default:

```
unsigned int SceneNode::getCategory() const
{
return Category::Scene;
}
```

Commanding (cont'd.)

- `getCategory()` can be overridden to return a specific category
- an aircraft belongs to the player if it is of type Eagle, and that it is an enemy otherwise:

```
unsigned int Aircraft::getCategory() const
{
  switch (mType)
  {
      case Eagle:
            return Category::PlayerAircraft;
      default:
            return Category::EnemyAircraft;
  }
}
```

Command struct Revisited

- we give our Command class another member variable that stores the recipients of the command in a category:

```
struct Command
{
   Command();
   std::function<void(SceneNode&, sf::Time)> action;
   unsigned int category;
};
```

- The default constructor initializes the category to Category::None. By assigning a different value to it, we can specify exactly who receives the command. If we want a command to be executed for all airplanes except the player's one, the category can be set accordingly:

```
Command command;
command.action = ...;
command.category = Category::AlliedAircraft
| Category::EnemyAircraft;
```

Command Execution

- Commands are passed to the scene graph
- Inside, they are distributed to all scene nodes with the corresponding game objects
- Each scene node is responsible for forwarding a command to its children
- SceneNode::onCommand() is called everytime a command is passed to the scene graph

```
void SceneNode::onCommand(const Command& command, sf::Time dt)
{  //check if the current scene node is a receiver of the command
   if (command.category & getCategory())
        command.action(*this, dt);

   FOREACH(Ptr& child, mChildren)
        child->onCommand(command, dt);
}
```

Command Queues

- A way to transport commands to the world and the scene graph
- A class that is a very thin wrapper around a queue of commands

```
class CommandQueue
{
public:
   void push(const Command& command);
   Command pop();
   bool isEmpty() const;

private:
   std::queue<Command> mQueue;
};
```

Command Queues (cont'd.)

- The `World` class holds an instance of `CommandQueue`:

```
void World::update(sf::Time dt)
{
    ...

    // Forward commands to the scene graph
    while (!mCommandQueue.isEmpty())
      mSceneGraph.onCommand(mCommandQueue.pop(), dt);

    // Regular update step
    mSceneGraph.update(dt);
}

CommandQueue& World::getCommandQueue()
{
    return mCommandQueue;
}
```

Player and Input

- Together now we're going to look at how the player's input is handled
- We will look at the following:
  - The `Player` class
  - The `processInput` function from `Game`