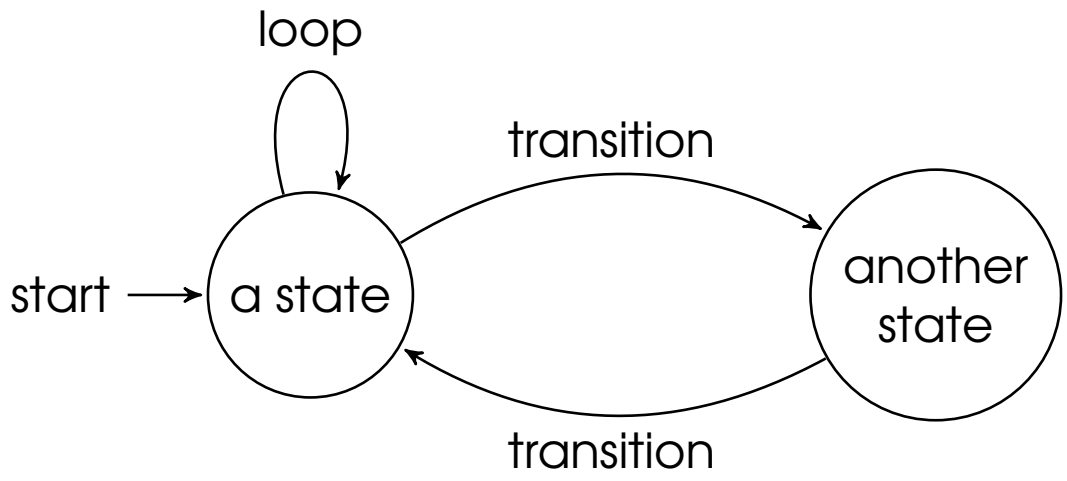
COMP110: Principles of Computing

Finite State Machines

Finite state machines

- A **finite state machine (FSM)** consists of:
  - A set of **states**; and
  - **Transitions** between states
- At any given time, the FSM is in a **single state**
- **Inputs** or **events** can cause the FSM to transition to a different state

State transition diagrams

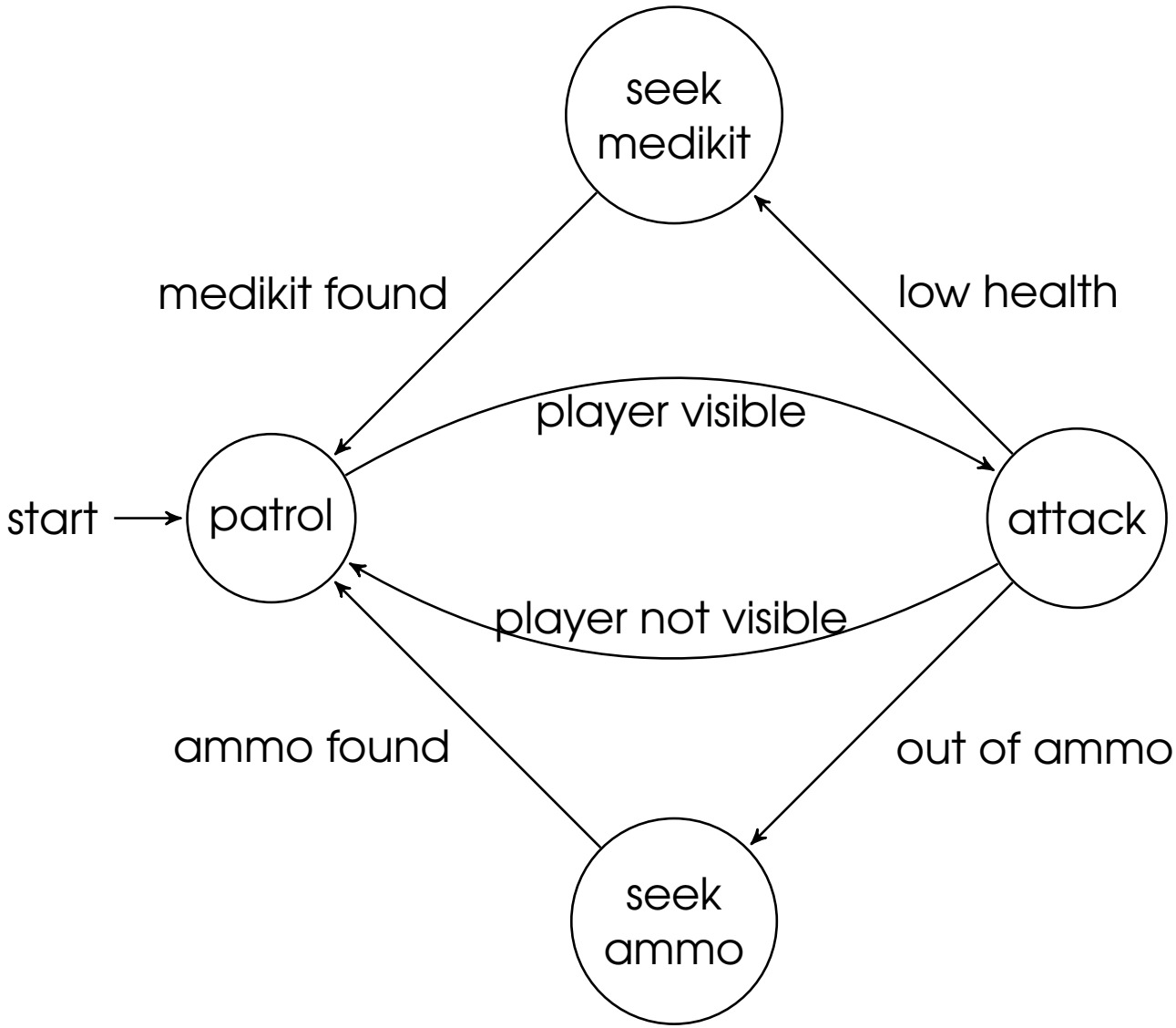


- FSMs are often drawn as **state transition diagrams**
- Reminiscent of **flowcharts** and certain types of **UML diagram**

FSMs for AI behaviour

The next slide shows a simple FSM for the following AI behaviour, for an enemy NPC in a shooter game:

- By default, patrol (e.g. along a preset route)
- If the player is spotted, attack them
- If the player is no longer visible, resume patrolling
- If you are low on health, run away and find a medikit. Then resume patrolling
- If you are low on ammo, run away and find ammo. Then resume patrolling


seek
medikit
medikit found
low health
player visible
start
patrol
attack
player not visible
ammo found
out of ammo
seek
ammo

Activity

- In your **COMP150 groups**, on the **whiteboards**
- **Extend** the FSM on the previous slide to produce **more realistic behaviour**
- Add at least **two** states, and associated transitions
- **5 minutes**...

Other uses of FSMs

As well as AI behaviours, FSMs may also be used for:

- UI menu systems
- Dialogue trees
- Token parsing
- ...

Activity

- In your **COMP150 groups**, on the **whiteboards**
- **Draw** an FSM for the overall UI structure of a typical game
- Include **at least** the following states, and appropriate transitions:
  - Main menu
  - Options menu
  - Game screen
  - Pause menu
  - Exit
- **5 minutes**...

FSMs in C++: Option 1

Use an **enum** to represent the states:

```
enum class SoldierState
{
    Patrol, Attack, SeekMedikit, SeekAmmo
};
```

Store the **current state** as a field, initialised to the start state:

```
SoldierState currentState = SoldierState::Patrol;
```

Use a **switch** statement to implement the FSM itself:

```
void Soldier::update()
{
    switch (currentState)
    {
    case SoldierState::Attack:
        attack();

        if (!isPlayerVisible())
            currentState = NPCState::Patrol;
        else if (health < LOW_HEALTH_THRESHOLD)
            currentState = NPCState::SeekMedikit;
        else if (ammo == 0)
            currentState = NPCState::SeekAmmo;
        break;
    // ...
    }
}
```

FSMs in C++: Option 2

Use **classes** to represent the states:

```
class SoldierState
{
public:
    virtual void update(Soldier& soldier) = 0;
};

class SoldierAttackState : public SoldierState
{
public:
    virtual void update(Soldier& soldier);
};
```

Implementing FSMs: Option 2

Store the current state as a (pointer to an) **instance** of a state class:

```
std::shared_ptr<SoldierState> currentState
    = std::make_shared<SoldierPatrolState>();
```

Use **virtual method overriding** to implement behaviour:

```
void SoldierAttackState::update(Soldier& soldier)
{
    soldier.attack();

    if (!soldier.isPlayerVisible())
        soldier.currentState
            = std::make_shared<SoldierPatrolState>();
    else if (soldier.health < LOW_HEALTH_THRESHOLD)
        soldier.currentState
            = std::make_shared<SoldierSeekMedikit>();
    else if (soldier.ammo == 0)
        soldier.currentState
            = std::make_shared<SoldierSeekAmmo>();
}
```

Beyond FSMs

Some topics for you to research, for when plain old FSMs aren't enough...

- Hierarchical FSMs
- Nested FSMs
- Stack-based FSMs
- Behaviour trees
- Hierarchical task networks
- ...

Summary

- Finite state machines (FSMs) can be used to model systems which have **one state** at any given time, and where inputs or events trigger **transitions** between states
- In games, often used for **simple AI behaviours** and for handling transitions between **UI screens**
- Can be implemented using **enums** and **switch** statements, or using **classes** and **polymorphism**