AI - State Machines

Introduction

The artificial intelligence displayed in games is usually due to one of two things: knowing where to go (and the best way to get there), and making decisions based on the situation. If an AI opponent runs away to find a health pack when their health drops too low, it's because some code ran an **if** statement on a threshold value somewhere, and then calculated a path to a part of the level expected to contain some heath packs.

In this part of the module, we'll be taking a look at the ways in which pathfinding, and decision making, can be integrated into a game engine, starting with a look into the idea of *finite state machines*.

Finite State Machines

The AI found in most games is pretty limited - it will do one of a number of different things, depending on a set of specific conditions within the game. To take classic arcade game *PacMan* as an example, the ghosts that chase the player always do the same things (in fact, they are so predictable that high-scoring players can accurately predict the ghost's actions); the ghosts wander around looking for the player, and chase after them if they can be seen, however if the player picks up a powerup, the ghosts will *always* run away until the powerup has expired.

While we certainly *could* model these changes of logic by having a set of **bools** and **if** statements, things quickly start to become unwieldy. In the above example we could probably get away with two bools for our ghosts - one for 'player is seen' and one for 'player has powerup', and from there build up some logic:

```
if(can see player) {
   ChasePlayer();
}
else if(player has powerup) {
   RunAway();
}
else {
   WanderMaze();
}
```

Simple PacMan example

This seems OK at first, but there's a problem - in that pseudocode above, the ghost will ignore the powerup state if it can see the player, and won't do the 'right' thing for the gameplay mechanics. Only if the 'player has powerup' state is checked first will the 'right' thing be done. But even by changing that, we'd not be modelling PacMan correctly - when a ghost is 'eaten' by the player, it has to return to the middle of the screen before it can chase the player once more. So we actually need *another* bool, and check for that one first, so that the ghost doesn't run away from the player once eaten. For this particular case, we could have an **enum** instead of a series of bools, giving us a mutually exclusive set of states that the ghost AI could be in:

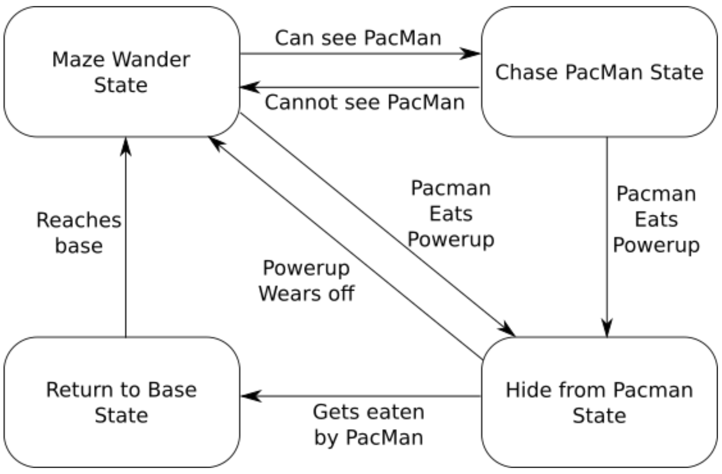
```
switch(ghostState) {
    case DEAD:      MoveToMiddle();break;
    case CHASE:     ChasePlayer();break;
    case POWERUP:   RunAway();break;
    case DEFAULT:   WanderMaze();break;
}
```

PacMan example with switch statements

That's probably OK for PacMan, and probably not a million miles away from how the logic was actually implemented (albeit probably in assembly). But what if we have states that can overlap, rather than being mutually exclusive? Maybe when the player has a powerup, the ghosts only run away when PacMan is nearby, and they otherwise still wander - now player distance is a metric, and it can modify multiple different states! That means more if statements, variables, and things that have to be in a specific order to do the 'correct' thing.

It's clear, then, that we need to in some way extend the logic of the AI in a more encapsulated way, so that every logical state that the AI can be in (and any data it needs to use as part of its functioning) is fully separated, allowing for a set of logic blocks that can be reasoned about in a clearer way.

Instead of representing each of the things an AI character can do as a single function, we can instead group together discrete logic blocks together as a *finite state machine*. A finite state machine is like a graph, where the individual nodes are 'states', or functions that in some way act upon an object. Connecting each node are edges or 'transitions', and each of these represents some decision or value, that if changed in a specific way, will lead to the current 'state' transitioning to another. So rather than just having a chunk of code consisting of tangled bools and if statements, we can instead model the specifics of our AI character, and when it should change its mind about doing specific things as a graph:



If we view our game logic like this, there's no possible ambiguity in what should happen, or when - the AI should enact the logic of exactly **one** of those states at a time, and should change to a different state if a specific transition condition is met. When designing behaviours for the AI agents in your games, it is often worthwhile trying to draw out your desired logic as a state machine diagram like the one above, as it gives you both a visual indication as to whether a behaviour should work, but also serve as a checklist of implemented functionality, and something to refer back to later when testing.

Modeling State Machines in code

While modelling the proposed logic of an AI character using a state machine diagram helps us decide what states we require to produce a desired effect, and which variables need to be considered to change which state should be executed, at some point this is going to have to be converted back into code, where errors could lead to incorrect, or at least undesirable, results. So how to turn the state machine back into code? Each state could be a simple function, with an AI then holding a pointer to the 'active' state function, which could then be modified by the state function itself:

```
void Ghost::GhostUpdate() {
   activeState(this, &activeState);
}

void Ghost::WanderState(Ghost& g, StateFunc* changeFunc) {
   g.position.x += rand01();
   g.position.y += rand01();
   if(PacmanGame::PowerpillActive()) {
      *changeFunc = RunAwayState;
   }
}

void Ghost::RunAwayState(Ghost& g, StateFunc* changeFunc) {
   //Make ghosts run away!
}
```

State machine as functions example

This works, but what if a particular state needs to store some addition state variables? Perhaps the wandering state has a speed value to modify the position by, or a counter that increments every frame, so that the ghost only changes direction every n frames? That's then another variable that needs to sit directly in the *Ghost* class of our example, where it could be misused, or modified by other states.

Instead, it may be better to have a *State* class, with a **virtual** *UpdateState* method that can be called every frame - subclasses can then be derived which override the *UpdateState* method, and which may keep some internal state as member variables; if they don't provide accessors, then they can have their state modified by any external code, and we can make stronger guarantees as to what the state will actually do in any frame.

We could then model an entire 'state machine' with a class that stores all of the states that it could be in, along with the state that is currently active:

```
class StateMachine {
   void UpdateMachine() {
      activeState->UpdateState();
   }
   State*activeState;
   vector<State*> allStates;
}
```

State Machine class pseudocode

If we are going to have use a class to represent a *state*, perhaps there should also be a class to represent a *transition*, too? This would allow us to fully encapsulate the concept of a state - no state knows about any other state, but we can still write logic inside a 'Transition' class to handle the movement from state A to state B:

```
class Transition {
   virtual bool ShouldTransition() {
   }
   State* GetState() {
      return newState;
   }
   State* oldState;
   State* newState;
}
```

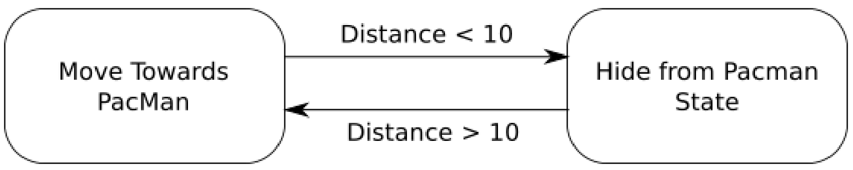
```
class RunOnPowerPillTransition : Transition {
   bool ShouldTransition() override {
      if(PacmanGame::PowerpillActive()) {
         return true;
      }
   }
}
```

Transition class pseudocode

By separating all of the workings of how an AI should operate or react to changes in the world out into discrete units of logic, it becomes easier to compose the logic in such a way that the correct thing will always happen, and making the process of debugging the AI in your game easier.

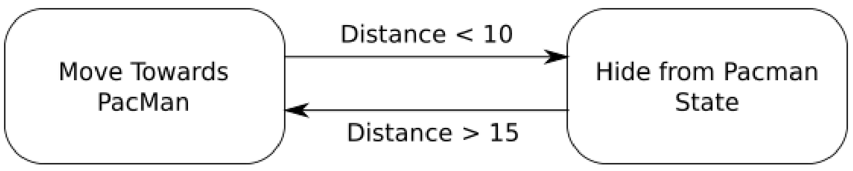
State oscillation and hysteresis

A state machine is 'dumb' in that it will do whatever it is you tell it to do, so without some careful planning of states and transitions, errors can still occur in your game AI. One such potential problem is that of *state oscillation*. Consider the following state machine to model the actions of a sneaky Ghost, who tries to follow the player, but who runs away if the player gets too close:



Seems sensible? You'd think so, but have a closer look. What happens when the Ghost gets to a distance of 10? According to our state machine behaviour, the Ghost will turn around and run away. Next frame, because the Ghost has ran away a small amount, the player is now 10 + a small amount away...causing the state machine to switch back to the 'follow the player' state. In other words, the states will begin to oscillate back and forth, and probably not result in the behaviour you'd like. In this particular case, as we're modelling a change in position, the Ghost would literally oscillate on the spot!

To solve this, we can introduce the concept of *hysteresis*. Broadly, hysteresis means that a system's behaviour is based not only on it's current state, but also taking into consideration its past state. While this *sounds* complicated, it can be actually quite simple for a state machine, as we're dealing with a limited number of states, and we know what possible changes should be made to those states. A simple example of state hysteresis in our Ghost example above would be to simply separate out the run away distance and follow distance:



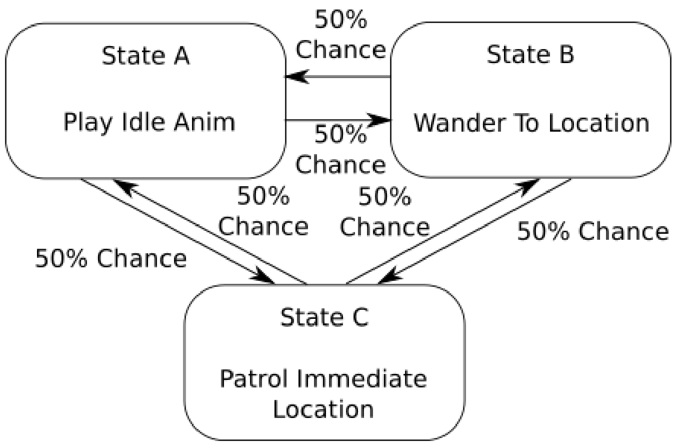
Even that small change is enough to stop the state rapidly oscillating, although the Ghost will eventually turn the other way. In this case our concept of hysteresis is to just consider the direction the distance value is moving in when transitioning to each state, and using it to accommodate the fact that if we're transitioning away from a state, we *probably* don't want to immediately transition back to it.

Another solution is to use *time* as part of our past state. This implementation could be as simple as a 'timeout' value per transition, that prevents the finite state machine from transitioning to a new state for a small amount of time, even if the conditions for a particular state would be met. An alternative could be for the transition to have a 'lead in' time, whereby that transition will only become active if its condition has been true for a given time period - both of these can produce broadly the same results, its just a case of which transition is 'in charge' of the delay (for timeouts it would be the transition that has been just activated, while for a 'lead in' time, it is the potential transition that *might* be activated).

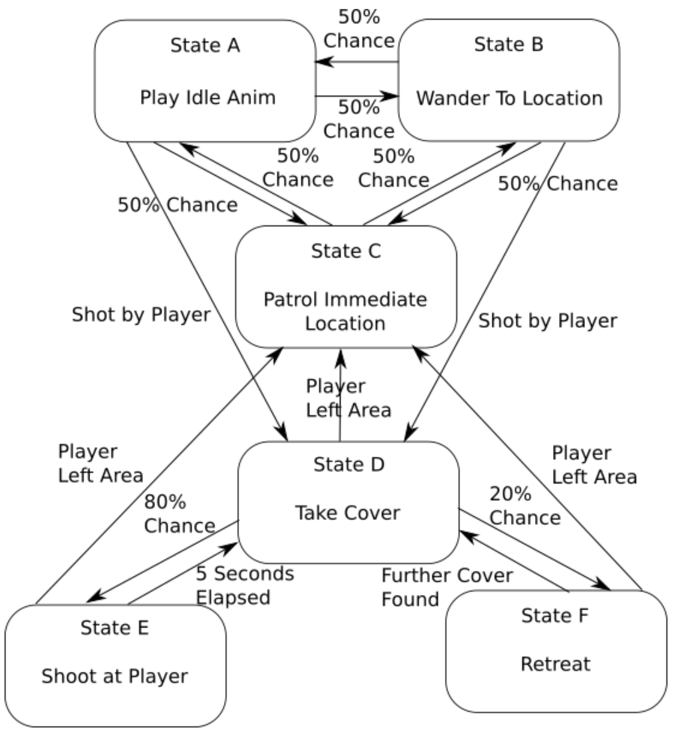
This can be quite a useful property in a finite state machine, as it can represent a reaction time to a particular event - maybe red ghosts are quicker to react to PacMan that blue ghosts - with the same state machine, a red ghost could have a shorter 'lead in' time than a blue ghost, making it harder to catch when running away, and also harder to run away from. This can produce an AI that feels a little bit more organic, and potentially more 'fair' for the player trying to beat the AI; most players can't immediately turn 180 degrees and land a perfect headshot in and FPS game, and so an AI probably shouldn't be able to, either.

Hierarchical State Machines

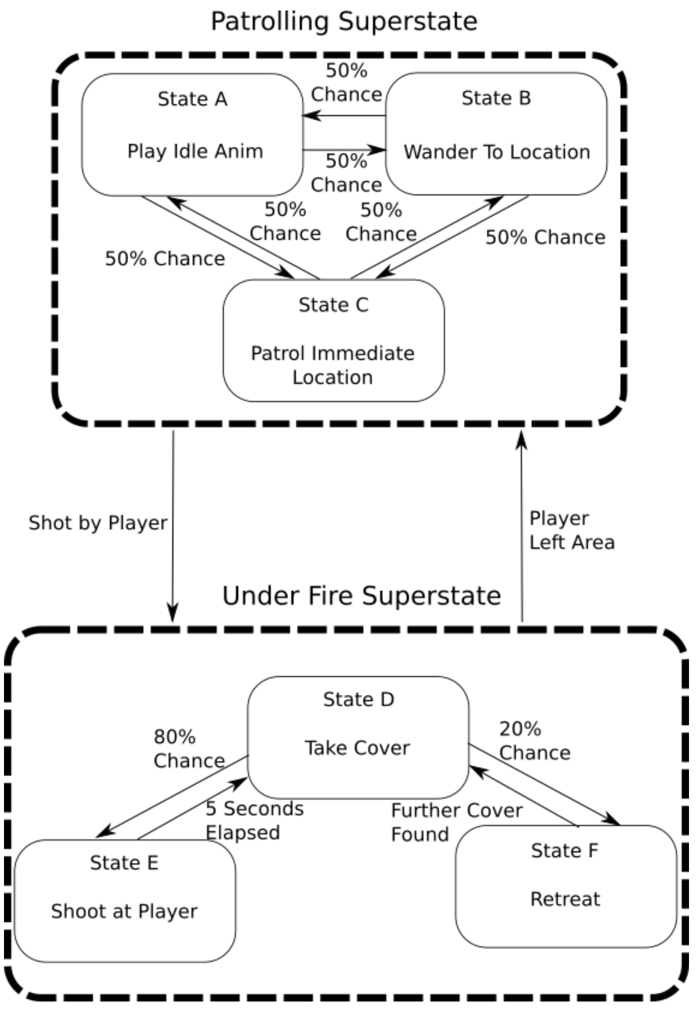
Sometimes the logic of a particular discrete state in an AI behaviours gets quite complex, too. Consider in an FPS game, where a patrolling AI opponent might wander around looking like its doing some important business:



On being shot at by the player, maybe the AI opponent might take cover and try and shoot back a the player, and if the player walks off, the opponent resumes their previous 'idling' behaviour. As a state machine, we might represent this newly enhanced behaviour like so:



That's a lot of transitions! The logic is starting to build up, and starting to look a little hard to follow again. More states and transitions means more C++ code is required, and more code means more potential mistakes. One potential solution to this is to replace the 'shot by player' set of states with *another* state machine! This is known as a *hierarchical* state machine, and in using one our AI opponent example might end up with a state machine like this:



If at any point one of the transitions of the 'shot by player' state are met (in this case, the player gets bored and walks off), the state transitions as normal back to the 'patrol' state. Otherwise, the logic actually enacted upon the AI opponent is part of the 'inner' child state machine. This allows for the same encapsulation of each piece of logic the AI might execute, but allows for us to reuse our state machine concept to keep the code simple - we end up with a lot of 'states', but each one has well defined inputs, outputs, and actions.

Tutorial Code

To get used to modelling state machines in code, we're going to take a quick look at how to make a 'standard' state machine as a set of extendable classes. The example won't do much, just increment some numbers which causes a transition to trigger in some states. But, in doing so, we'll be creating all of the underlying 'machinery' we need for our state machines, and from there allowing more complex states to be created for AI to use in its behaviour model.

StateMachine Class

To represent a state machine as a whole, we have a new class **StateMachine**, that should be added to the **CSC8503Common** project. This class will contain **std** containers to hold all of the states it can be in, and the transitions that control when to move from one state to another. Here's the code for the class:

```
namespace NCL {
   namespace CSC8503 {
      class State;   //Predeclare the classes we need
      class StateTransition;
      //Typedefs to make life easier!
      typedef std::multimap<State*, StateTransition*> TransitionContainer;
      typedef TransitionContainer::iterator TransitionIterator;

      class StateMachine   {
         public:
         StateMachine();
         ~StateMachine();

         void AddState(State* s);
         void AddTransition(StateTransition* t);

         void Update(float dt);

         protected:
         State*                activeState;
         std::vector<State*>   allStates;
         TransitionContainer   allTransitions;
      };
   }
}
```

State Machine class header

To start off with, this state machine class will be an empty box within which we can build up the general idea of what a stateful set of operations is, so its **constructor** doesn't have to do much, other than make sure that the *activeState* pointer starts off at a 'safe' **nullptr** value. As the machine is built up we'll be giving it some class instances to represent states and the transitions between them, so when the state machine is destroyed, we will have to **delete** these:

```
#include "StateMachine.h"
#include "State.h"
#include "StateTransition.h"
using namespace NCL::CSC8503;

StateMachine::StateMachine()  {
   activeState = nullptr;
}
```

State Machine class file

```cpp
StateMachine::~StateMachine() {
   for (auto& i : allStates) {
      delete i;
   }

   for (auto& i : allTransitions) {
      delete i.second;
   }
}
```

State Machine class file

To add a state, we just add the passed in parameter *s* to the *allStates* vector - if this is the first state added to the state machine, we'll assume it's the default starting state, too. Transitions are no different, except we're using a **multimap** to represent the directed graph nature of the state machine (that is, there can be multiple ways to transition away from a state), based on the source state as the map's *key*, and the transition itself as the *value*.

```cpp
void StateMachine::AddState(State* s) {
   allStates.emplace_back(s);
   if (activeState == nullptr) {
      activeState = s; //make this the default entry state!
   }
}

void StateMachine::AddTransition(StateTransition* t) {
   allTransitions.insert(std::make_pair(t->GetSourceState(), t));
}
```

State Machine class file

As you've probably found when using the STL, the combination of namespaces and templates can result in some particularly unwieldy variable names. To reduce this a little bit, on like 6 and 7 of the header file we made some *typedefs*, which let us create a new keyword that really maps onto a variable type - in this case were saying that a 'TransitionContainer' is really a **std::multimap** of a specific type, and similarly then defining the iterator type required to iterate through it on line 7. This saves us a little bit of space and time, and helps provide as much contextual information as possible when looking at code using these types.

Now, to update our state machine's behaviour, we just run *Update* on the active state, and then get the subset of transitions coming off that state - if one of them can transition to a new state, we get it and set our *activeState* pointer. The **StateMachine** class itself knows nothing about the workings of how a state works, what it is modelling, or how a transition decides when it should transition, making it nice and general purpose. We pass in a timestep to our *Update* method of the state machine itself, and pass it on to the actual states, so that they can if necessary create logic that operates in a correctly framerate independent manner. That's all there is to a finite state machine at the highest level - all of the actual implementation of a specific state machine can sit in the states themselves.

```cpp
void StateMachine::Update(float dt) {
   if (activeState) {
      activeState->Update(dt);
      //Get the transition set starting from this state node;
      std::pair<TransitionIterator, TransitionIterator> range =
      allTransitions.equal_range(activeState);
```

State Machine class file

```
      //Iterate through them all
      for (auto& i = range.first; i != range.second; ++i) {
         if (i->second->CanTransition()) { //some transition is true!
            State* newState    = i->second->GetDestinationState();
            activeState        = newState;
         }
      }
   }
}
```

State Machine class file

StateTransition Class

To represent a transition from one state to another, we're also going to make a **StateTransition** class, also in **CSC8503Common**, that holds pointers to some states, and which can also hold a function pointer. It will use this function to determine whether the machine can be transitioned away from its current state or not - we could do this with subclasses and virtual functions as well if we like, but this way keeps our code a little more concise, and as we'll see shortly, gives us practice with lambda functions! As with the state machine, the concept of a state transition is being kept away from the logic of any implementation of what should be checked to decide on when a transition should occur. This leads to a pretty sparse class:

```
#pragma once
#include <functional>
namespace NCL {
   namespace CSC8503 {
      class State;
      typedef std::function<bool()> StateTransitionFunction;
      class StateTransition   {
         public:
         StateTransition(State* source, State* dest,
         StateTransitionFunction f) {
            sourceState        = source;
            destinationState   = dest;
            function           = f;
         }
         bool CanTransition() const {
            return function();
         }
         State* GetDestinationState() const {return destinationState;}
         State* GetSourceState()      const {return sourceState;  }
         protected:
         State * sourceState;
         State * destinationState;
         StateTransitionFunction function;
      };
   }
}
```

StateTransition class header

State Class

All we can say about a state is that it in some way 'Updates' by running some logic, so our next class, **State**, again added to the **CSC8503Common** project, ends up with not much in it at all beyond another pointer to a function, which just as with state transitions, is our entry point for adding custom game logic to the state machine.

```
#pragma once
#include <functional>

namespace NCL {
   namespace CSC8503 {
      typedef std::function<void(float)> StateUpdateFunction;

      class  State      {
         public:
         State() {}
         State(StateUpdateFunction someFunc) {
            func     = someFunc;
         }
         void Update(float dt)  {
            if (func != nullptr) {
               func(dt);
            }
         }
         protected:
         StateUpdateFunction func;
      };
   }
}
```

State class header

Between this, the **StateTransition** class, and the **StateMachine** class itself, we don't need anything else to represent the *concept* of a state machine, but to create some concrete implementations of a state machine, we're going to have to inject some functions into our newly created classes.

Both the **StateTransition** and **State** classes are going to use **std::functions** to represent their function pointer objects. Even with this more modern way of representing a function pointer, the syntax is a little tricky to get your head around, so in both cases a C++ **typedef** is being used to create a custom type that represents a function of a specific signature. For the **StateTransition**, it is a function that takes in no parameters, but can return a **bool**, and for **States** it is a function that doesn't return anything, but takes in a **float** (representing our frame's timestep). By creating this as a **typedef**, it gets a little easier to look at the code and understand that it is using a function of a specific type, without having to keep looking at the long definition of a **std::function**.

Main function

To test our idea of a state machine, we're actually going to make a very simple one via a simple function call in our **main** function. Above the **main** function somewhere in the *Main.cpp* file, add the new function *TestStateMachine*, and start it off by defining a new instance of a **StateMachine**, and a single integer, which we will use to control the states and transitions:

```
void TestStateMachine() {
   StateMachine* testMachine = new StateMachine();
   int data = 0;
```

TestStateMachine function

```
    State* A = new State([&](float dt)->void
        {
            std::cout << "I'm in state A!\n";
            data++;
        }
    );

    State* B = new State([&](float dt)->void
        {
            std::cout << "I'm in state B!\n";
            data--;
        }
    );
```

TestStateMachine function

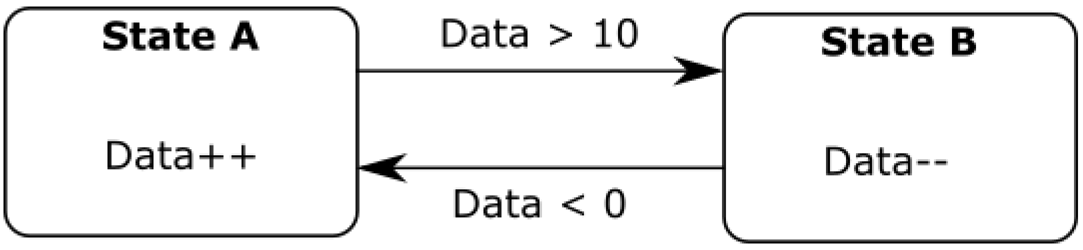
This creates two new instances of our **State** class, but also gives them some logic to operate on, via the creation of a lambda function. The square brackets starts off our function, which will take in the timestep (and so has a **float** parameter), but doesn't return anything. All the functions do for now is print to the console, and increment an integer, captured in the lambda by using an ampersand in the brackets that define a local lambda function.

Next we're going to do pretty much the same thing in defining a couple of state transitions. The functions called by our transitions must return a boolean value, to tell our state machine whether it can transition to a different state or not. We're again using the lambda brackets are defined such that they can capture values by reference - this is because we're going to determine whether to change state or not based on whether the integer defined earlier has gotten greater than 10, or less than 0, and use this to decide when to flip state. Without the ampersand, the lambda would not 'see' the integer, as it is in a different scope.

```
    StateTransition* stateAB = new StateTransition(A,B,[&](void)->bool
        {
            return data > 10;
        }
    );
    StateTransition* stateBA = new StateTransition(B,A,[&](void)->bool
        {
            return data < 0;
        }
    );
```

TestStateMachine function

Great! Now we have the 'engine' of our finite state machine, and can build up game logic in a discrete manner. To get a feel for how to compose a state machine using these classes, we're going to create the very simple 2 state machine:



Two states, two transitions, and that's it! To do so, finish off the *TestStateMachine* as follows:

```
    testMachine->AddState(A);
    testMachine->AddState(B);
    testMachine->AddTransition(stateAB);
    testMachine->AddTransition(stateBA);

    for (int i = 0; i < 100; ++i) {
        testMachine->Update(1.0f);
    }
}
```

TestStateMachine function

By making use of the generic 'skeleton' classes of a state machine, a state, and a transition, we can quickly prototype our two-state machine in just a few lines of code. Each transition makes use of a pointer to our *data* variable, and each state manipulates this value in some way. Combined, they demonstrate how a generic set of classes can be turned towards specific tasks within our codebase. This also shows us how the state machine can interact with the outside world - its states can be conditioned based on any data, not just something held within the classes that define it. Since it relies on function pointers, it has no real concept of what data is being read from or written to, only that it can pass a timestep variable around, and receive a **bool** to tell it to switch a pointer.

If we run the code, nothing will change graphically in the scene, but in the console we should see a bunch of text output - while the state machine has state *a* active it will print out a message due to the override *Update* method, which will then be transitioned away from in the next iteration.

State machines as game logic

You might well be looking at the state machine example above, and wonder how it could be turned to a more practical purpose. In our **TutorialGame** class, we create one or more **GameObjects** to populate the world with, which so far have represented purely unintelligent objects - we can push them around and make them act via forces, but they don't in any way 'think'. To remedy this, we can create a subclass of a **GameObject** that *can* think, and in this example will use a state machine to do something. In the **GameTech** project, make a new class called **StateGameObject**, and declare its header file like so:

```
#pragma once
#include "..\CSC8503Common\GameObject.h"
namespace NCL {
    namespace CSC8503 {
        class StateMachine;
        class StateGameObject : public GameObject  {
            public:
            StateGameObject();
            ~StateGameObject();

            virtual void Update(float dt);

            protected:
            void MoveLeft(float dt);
            void MoveRight(float dt);

            StateMachine* stateMachine;
            float counter;
        };
    }
}
```

StateGameObject class header

You can probably guess what this object will do based on just its method names and variables - it's going to move left and right, and periodically switch between the two using a **StateMachine** instance, and the *counter* variable.

The state machine will be created in the **constructor** of our new class. One of the benefits of the state machine using lambda functions, is that gives us quite a bit of choice as to what functions will be called to manipulate the state machine. In the previous example, the state functions performed the logic themselves (outputting some text to the screen). In this case, we're going to let the object itself hold the unique logic in the *MoveLeft* and *MoveRight* methods. The **State** and **StateMachine** classes don't know what a **StateGameObject** is, so they can't call member functions themselves, but what we *can* do is wrap up the calling of the member functions in yet more lambda functions. For a shape moving left and right, this could be considered overkill, and we could in this case have replicated the actions of the **MoveLeft** and **MoveRight** methods without further member variables, but it does serve the purpose of demonstrating both the strength of our state machines, and of lambdas.

```
#include "StateGameObject.h"
#include "../CSC8503Common/StateTransition.h"
#include "../CSC8503Common/StateMachine.h"
#include "../CSC8503Common/State.h"

using namespace NCL;
using namespace CSC8503;

StateGameObject::StateGameObject() {
   counter = 0.0f;
   stateMachine = new StateMachine();

   State* stateA = new State([&](float dt)-> void
   {
      this->MoveLeft(dt);
   }
   );
   State* stateB = new State([&](float dt)-> void
   {
      this->MoveRight(dt);
   }
   );

   stateMachine->AddState(stateA);
   stateMachine->AddState(stateB);

   stateMachine->AddTransition(new StateTransition(stateA, stateB,
   [&]()-> bool
   {
      return this->counter > 3.0f;
   }
   ));

   stateMachine->AddTransition(new StateTransition(stateB, stateA,
   [&]()-> bool
   {
      return this->counter < 0.0f;
   }
   ));
}
```

StateGameObject class file

Each transition checks to see whether the counter variable has gotten too high or low, and if so, will cause the state machine to switch to the other state. That's all we need to do for our state machine, but to make it work, we must update it every game frame - in this case, were going to do this by the *Update* method, which we'll be calling from the **TutorialGame** class shortly. We also need to **delete** the state machine upon destruction of our object.

```
StateGameObject::~StateGameObject() {
   delete stateMachine;
}

void StateGameObject::Update(float dt) {
   stateMachine->Update(dt);
}
```

StateGameObject class file

```
void StateGameObject::MoveLeft(float dt) {
   GetPhysicsObject()->AddForce({ -100, 0, 0 });
   counter += dt;
}

void StateGameObject::MoveRight(float dt) {
   GetPhysicsObject()->AddForce({ 100, 0, 0 });
   counter -= dt;
}
```

StateGameObject class file

TutorialGame Class

To test our our new stateful object, we're going to create a new method, and a new variable in the protected section of the **TutorialGame** class declaration. Remember, you'll need to **#include** your new class header, and instantiate the new variable to **nullptr**!

```
   StateGameObject* AddStateObjectToWorld(const Vector3& position);
   StateGameObject* testStateObject;
```

StateGameObject class header

To create the contents of the **AddStateObjectToWorld** method, just copy the contents of the *AddBonusToWorld* method into it. Crucially though, we need to change the instantiation of the **GameObject** pointer from being a **new GameObject**, to being a **new StateGameObject**. Then, in the **InitWorld** method, we can do the following:

```
   testStateObject = AddStateObjectToWorld(Vector3(0, 10,0));
```

TutorialGame class file

Our last change is to go into the **UpdateGame** method, and add the following:

```
   if (testStateObject) {
      testStateObject->Update(dt);
   }
```

TutorialGame class file

Conclusion

Starting the program again you should now see that you have a new object in the world, that moves from side to side periodically, with fairly minimal changes to the **TutorialGame** class.

Finite state machines are incredibly useful tools when developing an AI in a game. First, a simple finite state diagram like the ones in this tutorial allow for a bit of visual sanity checking over what gameplay mechanics should do, and if there are any flaws or failure states in the behaviour being modelled. When it comes down to programming them, we get to separate the logic out in a way where it is immediately obvious what should be running at any particular time, and what data should be readable or writeable at any given point in the behaviours operation.

We can further compose hierarchical state machines, to handle nested sets of AI behaviours, in a way that allows for easy transition between one and another, in a way that still follows a preset, well-defined set of rules. The examples shown here aren't particularly expressive, but the complex AI of a 'real' game could be built up of 10s of potential states to be in, with many rules to govern what should happen at any particular point in time. As code gets more complex, it gets harder to reason about, and harder to debug, so being able to break it down into specific functions can be of great help in creating complex AI interactions. Beyond this, the 'classful' manner of state machine construction investigated here should provide some insight into how game engines like Unity allow for a visual representation of what an AI is doing frame by frame, and how it can be composed graphically via a node graph.

Further Work

1) The state machine example in this tutorial is enough to get you started with making more complex code interactions that affect your game world as a whole. You might like to try making a **StateMachine** instance in the **TutorialGame** class that has a set of states that move a physics body around the world in a pattern, with each state checking the distance of the body to a predefined target, adding forces to move towards it, and transitioning to the next state when the distance is less than a threshold value.

2) You could try modifying the above state machine to additionally check to see if a specific other physics body is close by, and if so, transition to a state that applies forces to move towards it, instead of the predefined path. If the 'other' physics body is being moved around by the mouse / keyboard, it should feel like the state machine body is patrolling the world, and chasing the 'player'.

3) You can further refine this behaviour by coding a *hierarchical* state machine, by having a subclass of **State** that itself holds a **StateMachine**, and updates it in its own *Update* method.

4) As your game world gets larger, with more and more custom logic, you might find it beneficial to give the **GameObject** class its own Update virtual function, and call it in the **GameWorld::Update** method, which currently takes in the timestep value, but doesn't do an awful lot with it. This would allow you to write subclasses to encapsulate any of the object construction or logic required for your game, whether it uses state machines, or its own custom logic just within its overridden *Update* method.