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:

```
1 if(can see player) {
2    ChasePlayer();
3 }
4 else if(player has powerup) {
5    RunAway();
6 }
7 else {
8    WanderMaze();
9 }
```

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:

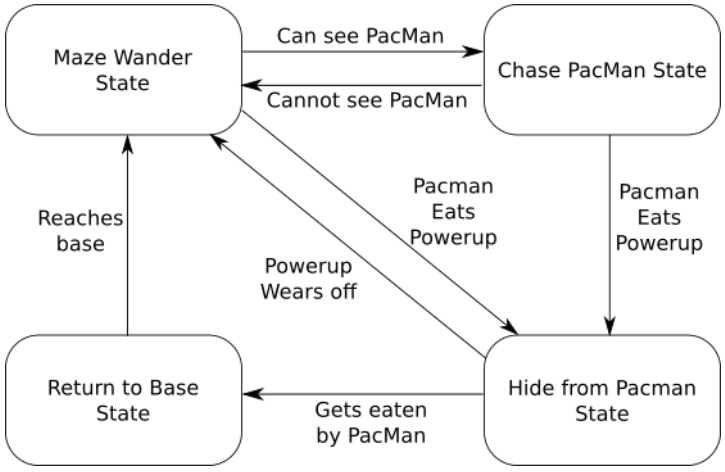
```
switch(ghostState) {
    case DEAD:      MoveToMiddle();break;
    case CHASE:     ChasePlayer();break;
    case POWERUP:   RunAway();break;
    case DEFAULT:   WanderMaze();break;
}
```

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;
}
```

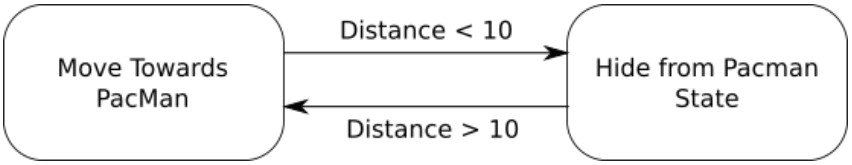
```
class RunOnPowerPillTransition : Transition {
    bool ShouldTransition() override {
        if(PacmanGame::PowerpillActive()) {
            return true;
        }
    }
}
```

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.

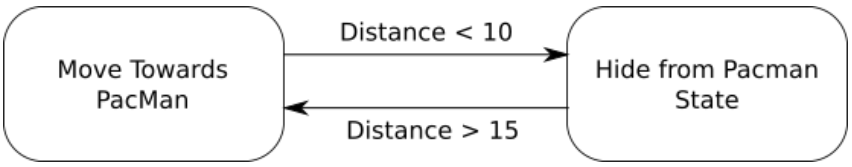
State oscillation and hysteresis

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.

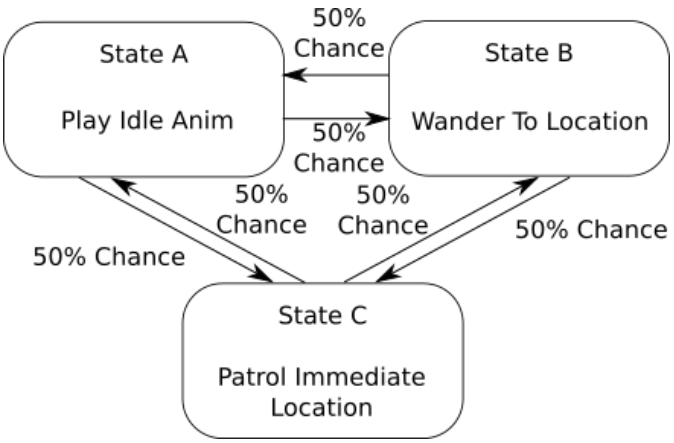
Another solution is to use *time* as part of our past state. This implementation could be as simple as a 'timeout' value per transition, that prevents the finite state machine from transitioning to a new state for a small amount of time, even if the conditions for a particular state would be met. An 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).

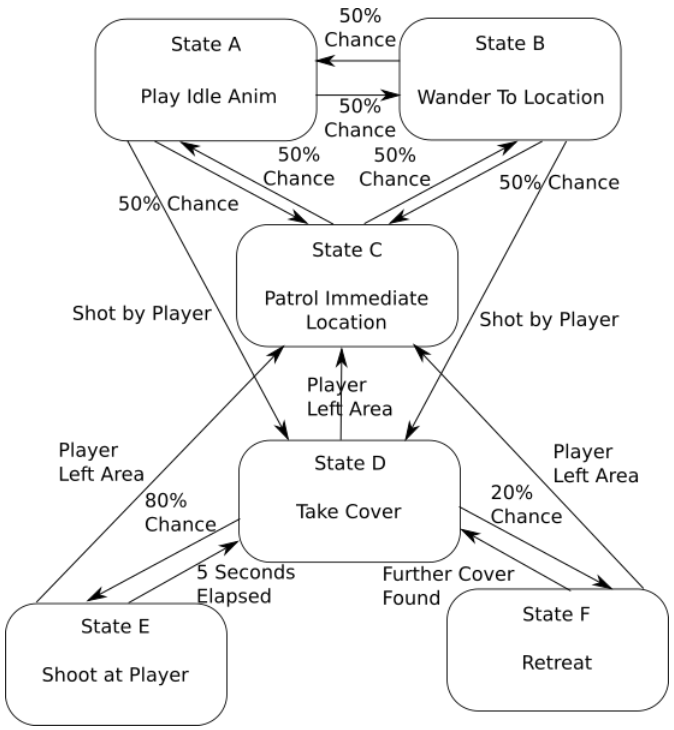
This can be quite a useful property in a finite state machine, as it can represent a reaction time to a particular event - maybe red ghosts are quicker to react to PacMan that blue ghosts - with the same state machine, a red ghost could have a shorter 'lead in' time than a blue ghost, making it harder 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

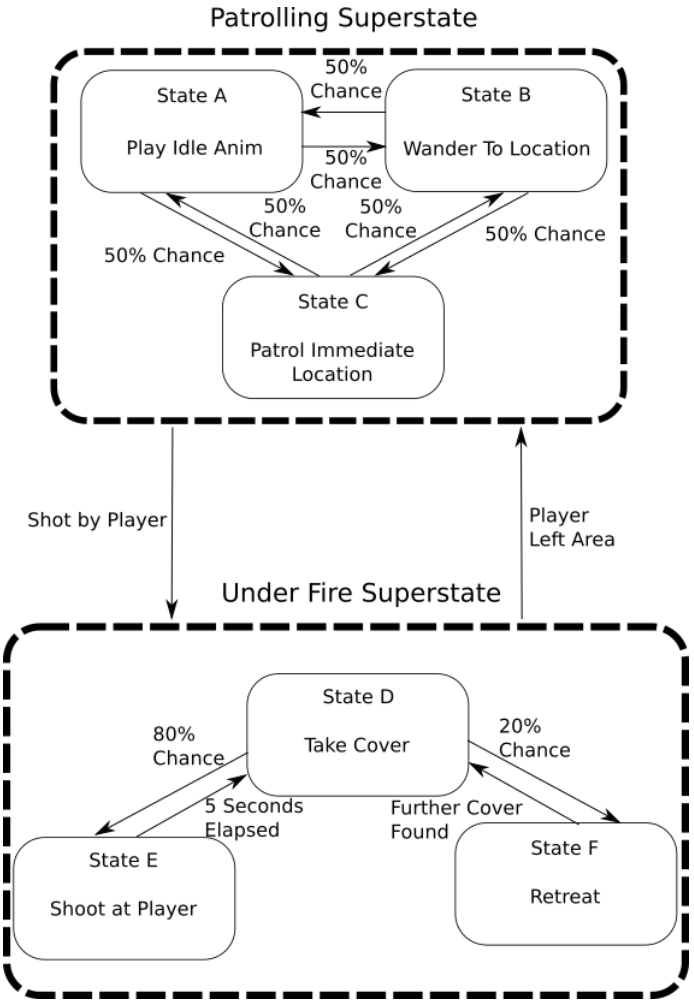
Sometimes the logic of a particular discrete state in an AI behaviours gets quite complex, too. Consider in an FPS game, where a patrolling AI opponent might wander around looking like its doing some important business:



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:



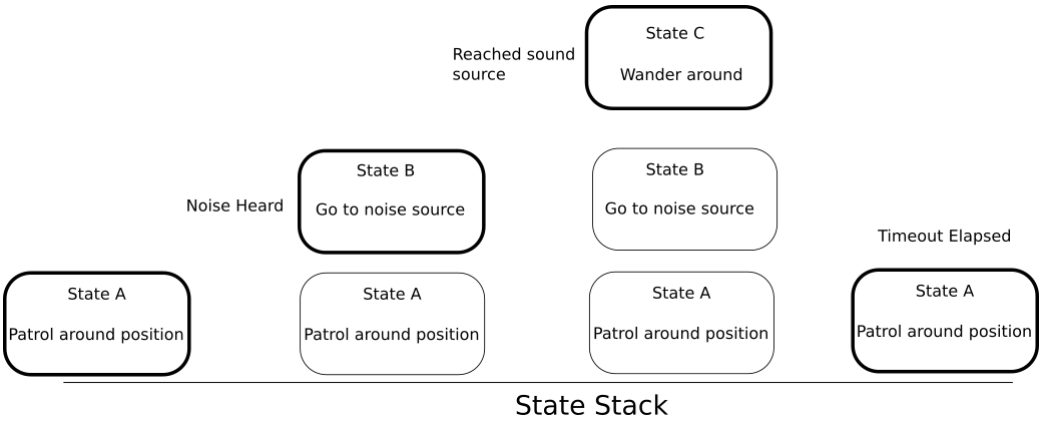
If at any point one of the transitions of the 'shot by player' state are met (in this case, the player gets bored and walks off), the state transitions as normal back to the 'patrol' state. Otherwise, the logic actually enacted upon the AI opponent is part of the 'inner' child state machine. This allows 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.

Pushdown Automata

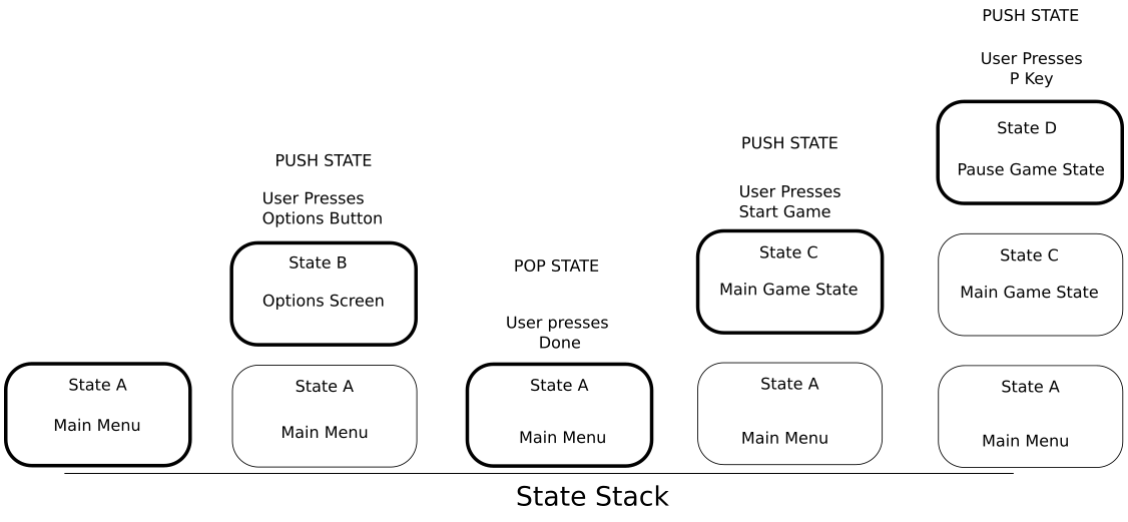
In our model of a state machine so far, there's only ever one 'copy' of each state per object, it just may or may not be executed in a particular game update depending on whether it has been transitioned to or not. Sometimes, we might actually *want* to have multiple copies of a state. For example, we might want an AI character to temporarily suspend doing their current task and go and do something else for a bit (maybe stop patrolling their level area and go and investigate a noise) before going back to whatever they were doing previously. Maybe that 'something else' will lead to additional 'side jobs' that will eventually be completed. For some combinations of tasks, we *could* model this with a standard finite state machine (Patrol $->$ investigate noise $->$ wander about $->$ Patrol), but in doing so we may lose some information - in this example there's nothing to make the AI character go back to where they used to be patrolling before hearing the noise, and so unless there's some hard-coded location data, they would instead start patrolling wherever they'd ended up after investigating a noise.

A common method of storing state information temporarily is to use a *stack*, a 'first in, last out' data structure. Even if you've never programmed or directly used a 'stack' object in C++, if you've ever called a function, you've indirectly used a stack to store the state of the CPU registers when calling a function (data is *pushed* onto the stack), which can then be restored after the function has returned (the data is *popped* off the stack). Even if that function calls a stream of additional functions, as long as we have enough stack space to keep pushing states, we can eventually pop them all off and get back to where we started.

We can combine the features of a state machine and a stack to form a *pushdown automata*. Rather than transitioning from state to state (with each state being a single 'instance' in memory), we can move to a new state by pushing a new instance of a state onto a stack - only the state at the top of the stack will be executed/updated per game tick. New states can later be pushed on, but we can always get back to our initial state by progressively popping states off. What determines whether a state is pushed or popped is a transition just as before, its just now there's a strict ordering, and a strict history of states - we can push any new type of state we want onto the stack, but we can only ever pop back to a previous state. As an example of this, we can model the patrolling and noise investigating behaviour:



Each of the 'Patrolling' states holds where the AI was when a new state was popped, and so maintains a history of what the AI was doing before a new decision had to be made, and so can resume that behaviour - perhaps on being 'popped' back to, a 'patrol' state immediately pushes a 'move to position' state onto the stack, and provides it with the position of where the AI was before investigating some noise, which will then pop itself off the automata stack when the AI is within a threshold distance of that position. By cutting up the actions an AI may perform into discrete states, we can then compose quite expansive AI logic, with the strict rules for transitioning from state to state allowing what should happen with an AI at any point in time to be reasoned about.



The idea of pushdown automata has its uses beyond agent AI. Think about how the main menu of a game works - there'll be some buttons which open up other menus (to set the controls, or graphics options etc), and a big 'Start Game' button. Once the game's started, maybe the player will eventually trigger a cutscene where control is taken away temporarily to play a video, or maybe the player presses the pause button, bringing up a little UI that allows them to save/load/quit the game. Pressing quit returns us all the way back to the main menu again. Sound familiar? We can model this behaviour using a pushdown automata like the one above.

As with many of the concepts introduced over the course of this tutorial material, there's often more than one use for them, so even if you don't require the complex state behaviour that can be expressed by a pushdown automata, keep them in mind for other parts of your games.

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**. 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:

```
1  namespace NCL {
2     namespace CSC8503 {
3        class State;   //Predeclare the classes we need
4        class StateTransition;
5  //Typedefs to make life easier!
6  typedef std::multimap<State*, StateTransition*> TransitionContainer;
7        typedef TransitionContainer::iterator TransitionIterator;
8
9        class StateMachine   {
10       public:
11          StateMachine() {  activeState = nullptr;}
12          ~StateMachine() {}
13
14          void AddState(State* s);
15          void AddTransition(StateTransition* t);
16
17          void Update();
18
19       protected:
20          State*                activeState;
21          std::vector<State*>   allStates;
22          TransitionContainer   allTransitions;
23       };
24    }
25 }
```

State Machine class header

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, based on the source state as the map's *key*, and the transition itself as the *value*.

```
1  void StateMachine::AddState(State* s) {
2     allStates.emplace_back(s);
3     if (activeState == nullptr) {
4        activeState = s;
5     }
6  }
7
8  void StateMachine::AddTransition(StateTransition* t) {
9     allTransitions.insert(std::make_pair(t->GetSourceState(), t));
10 }
```

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 we can see 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.

```
11 void StateMachine::Update() {
12    if (activeState) {
13       activeState->Update();
14       //Get the transition set starting from this state node;
15       std::pair<TransitionIterator, TransitionIterator> range =
16          allTransitions.equal_range(activeState);
17       //Iterate through them all
18       for (auto& i = range.first; i != range.second; ++i) {
19          if (i->second->CanTransition()) { //some transition is true!
20             State* newState   = i->second->GetDestinationState();
21             activeState       = newState;
22          }
23       }
24    }
25 }
```

State Machine class file

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.

StateTransition Class

To represent a transition from one state to another, we're also going to make a **StateTransition** class, that holds pointers to some states, and which has a single **pure virtual** function **CanTransition**, that will return **true** if some internal condition is met, meaning that our state machine should transition to a new state 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:

```
namespace NCL {
   namespace CSC8503 {
      class State;
      class StateTransition   {
      public:
         virtual bool CanTransition() const = 0;
         State* GetDestinationState() const {return destinationState;}
         State* GetSourceState()      const {return sourceState;  }
      protected:
         State * sourceState;
         State * destinationState;
      };
   }
}
```

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 class representing this ends up with not much in it at all beyond a **pure virtual** *Update* method:

```
namespace NCL {
   namespace CSC8503 {
      class State    {
      public:
         State();
         virtual ~State();
         virtual void Update() = 0; //Pure virtual base class
      };
   }
}
```

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 make some subclasses.

GenericTransition Class

Quite often the decision on whether to transition from one state to another in our state machines is based on a simple value comparison - if health < 20, enter 'run away state', if distance from enemy $>$ 10, enter sprint state, and so on. So that means we can actually model many common transitions by storing a value, and then performing a comparison check, as part of the *CanTransition* method of the **StateMachine** class. To model this behaviour, we're going to make a subclass of **StateTransition**, and use templates to allow it to compare one object against another, using a passed in function. This allows us to check versus an object's (imaginary!) *GetHealth*() or *GetPosition*() methods just as easily, with one method.

The class itself is fairly straightforward, but the use of templates makes it look a bit more awkward than it really is. The purpose of the class is to transition from one state to another, if a comparison between two particular values passes. To do this though, we actually need *two* templated types - If we wanted to compare a **PhysicsObject's** position against a position in the world, we'd need to store the **PhysicsObject**, and a **Vector3**, as even though we're extracting a **Vector3** to compare, we need to keep hold of the **PhysicsObject** in the meantime to know what to check. Our **GenericTransition** class starts as follows:

```
1       template <class T, class U>
2       class GenericTransition : public StateTransition   {
3       public:
4          typedef bool(*GenericTransitionFunc)(T, U);
5          GenericTransition(GenericTransitionFunc f,
6          T testData, U otherData, State* srcState, State* destState) :
7             dataA(testData), dataB(otherData)
8             func              = f;
9             sourceState       = srcState;    //
10            destinationState  = destState;
11         }
12         ~GenericTransition() {}
13
14         virtual bool CanTransition() const override{
15            if (func) {
16               return func(dataA, dataB);
17            }
18            return false;
19         }
20      protected:
21         GenericTransitionFunc  func;
22         T  dataA;
23         U  dataB;
```

GenericTransition class header

We store a couple of pieces of data (they might be the same type, there's nothing about a templated type that disallows two separate template types from being of the same type), and a function pointer that the *CanTransition* method uses to determine whether it's time to change from one state to another. The only interesting thing is that we must instantiate the *dataA* and *dataB* variables with an initialiser list (shown on line 7). Why? Our template type might be a *reference*, and therefore the templated variables *must* be filled in in this manner.

The final part of this class, back in the public section of the header, is a set of generic static functions that can perform simple comparisons - we don't *have* to use them (we could just pass in a lambda, instead), it does provide another example of being able to pass around function pointers. If we want, we can pass these functions in to the constructor of a *GenericTransition*, as they meet the function signature of the *func* variable, which was **typedef'd** on line 4.

```
24      public:
25         static bool GreaterThanTransition(T dataA, U dataB) {
26            return dataA > dataB;
27         }
28         static bool LessThanTransition(T dataA, U dataB) {
29            return dataA < dataB;
30         }
31         static bool EqualsTransition(T dataA, U dataB) {
32            return dataA == dataB;
33         }
34         static bool NotEqualsTransition(T dataA, U dataB) {
35            return dataA != dataB;
36         }
37      }
38   };    //end of namespace CSC8503
39 };    //end of namespace NCL
```

GenericTransition class header

GenericState Class

As well as creating whole subclasses to represent a state, we can take advantage of function variables and lambdas to create a 'generic' state that simply runs a function. This becomes harder to use if a particular state needs to keep lots of variables around, but we can easily plug basic functions into a state machine by creating a class that looks like the following:

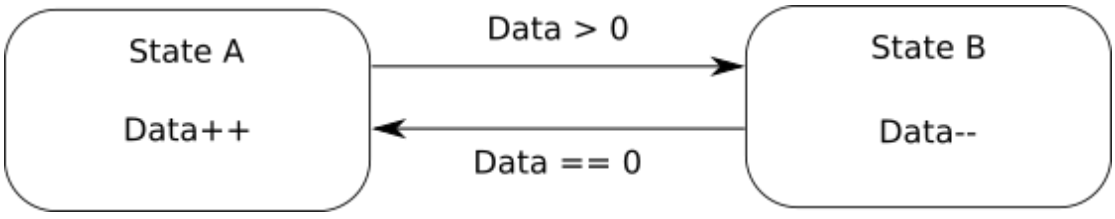
```
#pragma once

namespace NCL {
   namespace CSC8503 {
      typedef void(*StateFunc)(void*);

      class GenericState : public State      {
      public:
         GenericState(StateFunc someFunc, void* someData) {
            func     = someFunc;
            funcData = someData;
         }
         void Update() override {
            if (funcData != nullptr) {
               func(funcData);
            }
         }
      protected:
         StateFunc func;
         void* funcData;
      }; //end of GenericState class
   } //end of CSC8503 namespace
} //end of NCL namespace
```

GenericState Class

This **GenericState** class overrides the *Update* method of the **State** class, and uses it to call *another* function, with a single stored piece of state as a parameter. This piece of state is simply a **void** pointer, which the function will have to **cast** to something sensible to operate on - we'll see an instance of this in action shortly.

Main 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, we're going to just fill in a function called by the main function, *TestStateMachine* as follows:

```
1  void TestStateMachine() {
2      StateMachine* testMachine = new StateMachine();
3
4      int someData = 0;
5
6      StateFunc AFunc  = [](void* data) {
7          int* realData = (int*)data;
8          (*realData)++;
9          std::cout << "In State A!" << std::endl;
10     };
11     StateFunc BFunc  = [](void* data) {
12         int* realData = (int*)data;
13         (*realData)--;
14         std::cout << "In State B!" << std::endl;
15     };
16
17     GenericState* stateA = new GenericState(AFunc, (void*)&someData);
18     GenericState* stateB = new GenericState(BFunc, (void*)&someData);
19     testMachine->AddState(stateA);
20     testMachine->AddState(stateB);
```

TestStateMachine function

We're defining a couple of local functions as lambdas - one will take in a pointer and try and increment the value it points to (line 8), while the other decrements it (line 13). From these functions, and the integer on line 4, we can form a couple of **GenericStates**.

We can model the transitions between our simple machine's states using the **GenericTransition** class from earlier - this time templated such that each one takes in a reference to an integer (the variable from line 4), and an integer that will be used as a comparison value. We can pass in the *GreaterThanTransition* and *EqualsTransition* functions (templated the same way) to then perform the comparison.

```
21     GenericTransition<int&, int>* transitionA =
22         new GenericTransition<int&, int>(
23         GenericTransition<int&, int>::GreaterThanTransition,
24         someData, 10, stateA, stateB); //if greater than 10, A to B
25
26     GenericTransition<int&, int>* transitionB =
27         new GenericTransition<int&, int>(
28         GenericTransition<int&, int>::EqualsTransition,
29         someData, 0, stateB, stateA); //if equals 0, B to A
30
31     testMachine->AddTransition(transitionA);
32     testMachine->AddTransition(transitionB);
33
34     for (int i = 0; i < 100; ++i) {
35         testMachine->Update(); //run the state machine!
36     }
37     delete testmachine;
38 }//end of TestStateMachine function!
```

TestStateMachine function

Once the transitions and states are added to the machine, it's ready to run! As a test, we just run the *Update* for the *testMachine* 100 times - it should start off in *stateA* (as it was the first state added), and after 10 iterations (which should run the *AFunc* lambda), the checks of the transitions within the *StateMachine::Update* method should test against *transitionA*, and trigger the change into *stateB*. After 10 more iterations, the same mechanisms will trigger *transitionB*, to change back into *stateA*.

Conclusion

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, an integer increases, which will eventually cause transition *a* to trigger, resulting in state *b* becoming active. This state then decreases the integer back down until it hits 0, where another transition will trigger, causing the state to revert back to *a* again.

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. Beyond even this, we can use state machine behaviour to model interactions beyond the raw AI of our game - the menus in our games can be managed by state machines, and even which gameplay code should be used at a given time could all be encapsulated within a state machine. When some historical information is required (previous menus visited, previous tasks that have been temporarily paused), we can additionally make use of pushdown automata to give our state machines an additional stack of states, and their related required data.

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) There is the skeleton of a Pushdown automata inside the codebase download, made out of the **PushdownMachine** and **PushdownState** classes. You may wish to use these as the basis for some basic mechanics requiring a history of states. The 'main menu' for your coursework game is probably a good place to try this out, as a set of classes, each of which outputs some information to the screen, and may pause / restart the underlying physics or game mechanics of the program.