AI - Pushdown Automata

Introduction

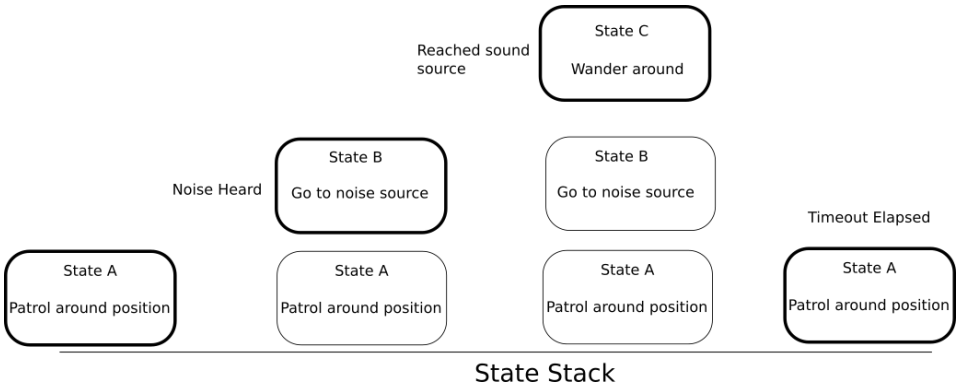
Finite state machines help us model the behaviour of the AI agents within our game. There's more to this idea of states than just AI, though - almost every part of a game can be thought of as a state, with a limited number of states that could be transitioned to. In this tutorial we'll take a look at a slightly different way of representing how to move from one state to another, via the use of a *pushdown automata*, a mechanism that allows a stack of states to be built up, giving our state machine an idea as to the history of what states have been executed.

Pushdown Automata

In our model of a state machine so far, a state has no real concept of what happened to lead to it being executed, or the wider change in world state that led up to the transitions being processed to cause it to become active. There's only ever one 'copy' of each state within the machine, 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 the function was called (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 just 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:

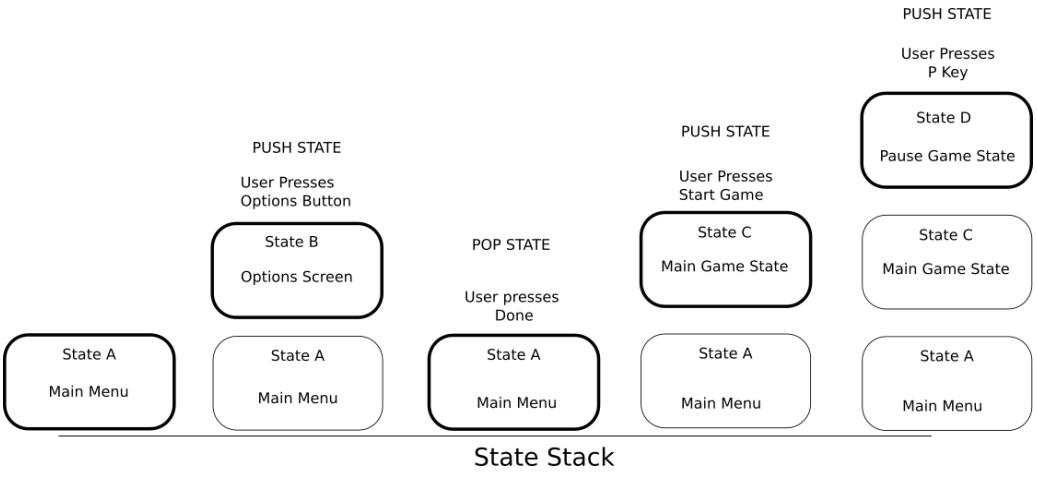


Each of the 'Patrolling' states holds where the AI was when a new state was popped, and so maintains a history of what the AI was doing before a new decision had to be made, and so can resume that behaviour - perhaps on being 'popped' back to a 'patrol' state immediately pushes a 'move to position' state 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.

Of course, some states might have a limited amount of time that they matter for - a noise that the AI heard 10 minutes ago might well have been forgotten in real-life, and may be irrelevant to whatever the AI is currently doing. This could be handled by a simple timer within a state, such that if a state is ever returned to after a long enough period, the state also pops itself off the stack, until eventually there's either nothing left on the stack, or a state with sufficient importance or time left in its timer becomes active. Another alternative could be to periodically 'reset' the stack, clearing out all of the states, and then pushing back a default initial state. In our AI example, this could be the simple 'patrolling' state, allowing the AI to eventually give up whatever sequence of events might have led it to move away from its path, and just go back to wandering its preset route.

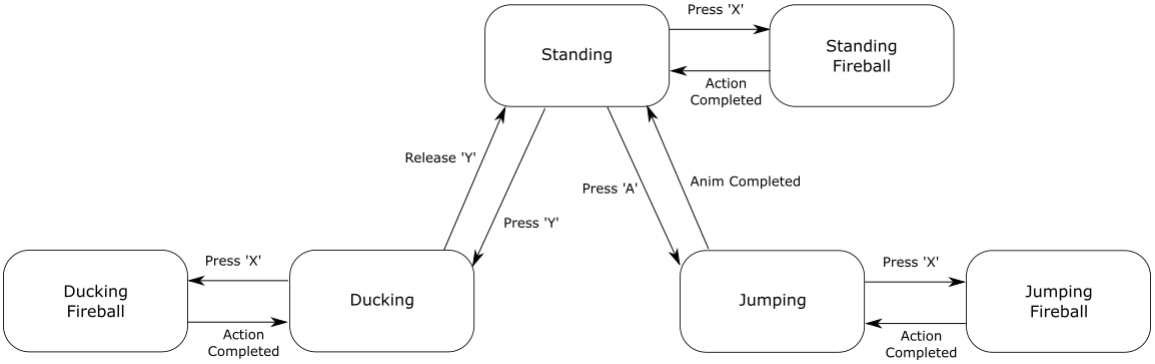
Pushdown Automata beyond AI

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:

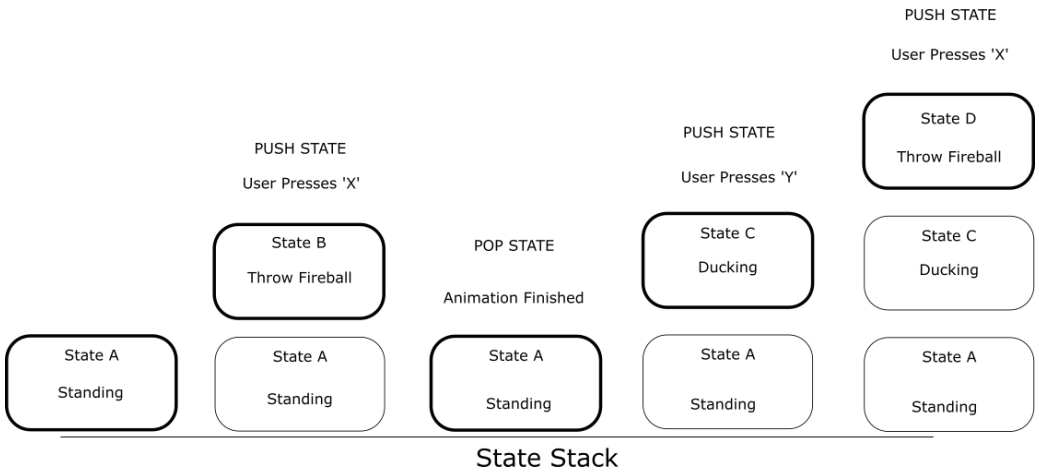


By splitting up the logic that we want to follow in our menu into discrete states, it becomes easier to manage the flow of user interaction. The pushdown automata intrinsically 'knows' that it should show a particular screen when exiting out of another, even if the logic of either of those machines knows nothing about the other. This increases the encapsulation of our game code, and makes it easier to add new functionality to it without breaking other parts.

Another useful use could be in the controlling of the player state, too. While the player character isn't an 'AI', the rules of the second-to-second gameplay might dictate that some actions can or cannot be performed due to the user's previous input. As an example of this, consider a fighting game, in the style of Street Fighter. A character in such a game may have a different subset of moves that can be achieved depending on whether they are standing, in the air, or ducking, but let's assume for now that they can always throw a fireball. If we imagine that the 'throw fireball' action in the game is a separate state, that handles the playing of the correct animation, spawning of the fireball object, and so on, then we hit a problem - how does this state know how to transition back to what the player was doing before executing the 'throw fireball' move? They could have been standing, ducking, or in the air after having jumped, and each of these might have different animations and move subsets. We *could* have *three* 'throw fireball' states to handle whether the player was in the air, ducking, or standing:



This is going to lead to some duplicate code, or the creation of even more functions for different states to call. Instead, we could make use of a pushdown automata to handle the fireball action:



Now When the 'throw fireball' state has completed its working, it can pop itself off the state stack, and return to doing whatever the player was doing beforehand, without needing any additional states or duplication of their workings. It's the same end functionality, but less states were required to make it operate effectively.

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

Tutorial Code

To get a feel for how a stack can be augmented to provide us with an enhanced means of running states, we are again going to make a generic set of classes to provide the base machinery of a pushdown automata. This time, rather than use lambda functions to enact the unique gameplay states, we're going to see how virtual functions can be used, providing a little more practice with polymorphism and when to create a new class.

PushdownState Class

Similarly to a 'normal' state machine, each state will be encapsulated within a class, that has a function that will be called by the machine every frame. This time around, it's a virtual function, and also has a pair of additional methods, that will be called when a state is 'put to sleep' by having another state popped onto the stack, or 'woken up' by becoming the top of the stack (either by being the first state, or by having a state popped off the stack).

As the state of a pushdown automata can be changed by the actions of it's 'active' state, the *OnUpdate* method of this class will return an enum to state whether this state should continue, be popped off the stack, or is indicating that a new state should be pushed onto the stack. For this latter reason, the *OnUpdate* method, as well as taking in the timestep as usual, also takes in a *pointer to a pointer* - this might sound a bit scary, but it will allow a **PushdownState** to instantiate whichever new state should be operated on by the state machine.

```
1  #pragma once
2
3  namespace NCL {
4     namespace CSC8503 {
5        class PushdownState  {
6           public:
7           enum PushdownResult {
8              Push, Pop, NoChange
9           };
10          PushdownState()             {}
11          virtual ~PushdownState()    {}
12
13          virtual PushdownResult OnUpdate(float dt,
14             PushdownState** pushFunc) = 0;
15          virtual void OnAwake() {}
16          virtual void OnSleep() {}
17       };
18    }
19 }
```

PushdownState class header

PushdownMachine Class

For the central class that represents our pushdown automata, we have a very simple setup - a **std::stack**, and a couple of pointers to the newly created **PushdownState** to represent the starting state, and the current state being executed. As with the state machines in the previous tutorial, there's an *Update* method to drive the whole thing.

```
#pragma once
#include <stack>

namespace NCL {
   namespace CSC8503 {
      class PushdownState;

      class PushdownMachine   {
         public:
         PushdownMachine(PushdownState* initialState) {
            this->initialState = initialState;
         }
         ~PushdownMachine() {}

         bool Update(float dt);

         protected:
         PushdownState* activeState;
         PushdownState* initialState;

         std::stack<PushdownState*> stateStack;
      };
   }
}
```

PushdownMachine class header

There's only one method to the **PushdownMachine** class, *Update*. It needs to call *OnUpdate* on its *activeState*, and react to the **PushdownResult** enum that it returns. Here's where we see why the *OnUpdate* method of the **PushdownState** class takes a pointer to a pointer - on line 7 we can see a pointer to a new **PushdownState**; the *OnUpdate* method will then receive a pointer to that variable, and so can, if necessary, instantiate a new **PushdownState** and place a pointer to it in the *newState* variable.

```
#include "PushdownMachine.h"
#include "PushdownState.h"
using namespace NCL::CSC8503;

bool PushdownMachine::Update(float dt) {
   if (activeState) {
      PushdownState* newState = nullptr;
      PushdownState::PushdownResult result
         = activeState->OnUpdate(dt, &newState);

      switch (result) {
         case PushdownState::Pop: {
            activeState->OnSleep();
            delete activeState;
            stateStack.pop();
            if (stateStack.empty()) {
               return false;
            }
            else {
               activeState = stateStack.top();
               activeState->OnAwake();
            }
         }break;
```

PushdownMachine class file

```
24            case PushdownState::Push: {
25               activeState->OnSleep();
26
27               stateStack.push(newState);
28               activeState = newState;
29               activeState->OnAwake();
30            }break;
31         }
32      }
33      else {
34         stateStack.push(initialState);
35         activeState = initialState;
36         activeState->OnAwake();
37      }
38      return true;
39   }
```

PushdownMachine class file

Deciding what the automata should do is determined by switching against the **enum** that the *OnUpdate* method returns. If the *OnUpdate* method returns **Pop**, we can assume that it has decided it is finished with whatever actions it was doing, and it can therefore be popped off the state stack (line 15). If there's a state left on the stack, we can let it know it has become active by calling its virtual **OnAwake** method, and then make it the *activeState*, ready for updating in the next execution of the automata. The else on line 33 is for the first execution of the automata, and will set up the machine ready for running in the next frame; it's done here and not in the constructor as there might be some need to wait for the game to actually be up and running in the virtual **OnAwake** method, which wouldn't necessarily be the case when the automata is first constructed.

Creating a Pushdown Automata

To see the state stack in action, we're going to create a *very* simple game. It will have an intro state, a main game state, and a pause screen state. This might sound like it's going to be a lot of work, but these 'states' aren't going to really have any game logic, or even any graphics, just some text printed to the game console. In the main file of your CSC8503 project, add the following three classes and function above the **main** function.

Every single player game should have the ability to pause the on-screen action, and our test game is no exception! This is represented as a subclass of a **PushdownState**, that overrides the *OnUpdate* and *OnAwake* methods. If we press the U key, the 'game' will be unpaused, by having the pause state inform the automata to pop itself off the state stack (line 5), otherwise it'll wait (line 7), with players being informed of the key to press when the state is first enacted, via it's overridden *OnAwake* method (line 9).

```
1  class PauseScreen : public PushdownState {
2     PushdownResult OnUpdate(float dt,
3           PushdownState** newState) override {
4        if (Window::GetKeyboard()->KeyPressed(KeyboardKeys::U)) {
5           return PushdownResult::Pop;
6        }
7        return PushdownResult::NoChange;
8     }
9     void OnAwake() override {
10       std::cout << "Press U to unpause game!\n";
11    }
12 };
```

main file

Now for the game class. It's not really a game, just a rough facsimile of one, that will randomly increase a number every frame (line 31), and periodically inform the player of how large this number is (line 17). If the player presses P, then the game will be paused by the method returning a Push enum, and using the pointer to a pointer to instantiate a new **PauseScreen** class. If instead they press F1, then the player quits the 'game' and goes back to the start screen, by telling the automata to pop the state (line 29). Otherwise, the game keeps going, increasing a number, and telling the automata that the state should continue (line 34).

```
13 class GameScreen : public PushdownState {
14    PushdownResult OnUpdate(float dt,
15          PushdownState** newState) override {
16       pauseReminder -= dt;
17       if (pauseReminder < 0) {
18          std::cout << "Coins mined: " << coinsMined << "\n";
19          std::cout << "Press P to pause game,
20                        or F1 to return to main menu!\n";
21          pauseReminder += 1.0f;
22       }
23       if (Window::GetKeyboard()->KeyDown(KeyboardKeys::P)) {
24          *newState = new PauseScreen();
25          return  PushdownResult::Push;
26       }
27       if (Window::GetKeyboard()->KeyDown(KeyboardKeys::F1)) {
28          std::cout << "Returning to main menu!\n";
29          return PushdownResult::Pop;
30       }
31       if (rand() % 7 == 0) {
32          coinsMined++;
33       }
34       return PushdownResult::NoChange;
35    };
36    void OnAwake() override {
37       std::cout << "Preparing to mine coins!\n";
38    }
39    protected:
40    int   coinsMined     = 0;
41    float pauseReminder  = 1;
42 };
```

main file

The last class we need is an 'intro' screen to welcome the player to this fantastic game. In its *OnUpdate* method it simply watches for which key has been pressed, and either pops or pushes a state, much as with the **GameScreen** class.

```
43 class IntroScreen : public PushdownState {
44    PushdownResult OnUpdate(float dt,
45          PushdownState** newState) override {
46       if (Window::GetKeyboard()->KeyPressed(KeyboardKeys::SPACE)) {
47          *newState = new GameScreen();
48          return PushdownResult::Push;
49       }
50       if (Window::GetKeyboard()->KeyPressed(KeyboardKeys::ESCAPE)) {
51          return PushdownResult::Pop;
52       }
53       return PushdownResult::NoChange;
54    };
```

main file

```
55      void OnAwake() override {
56          std::cout << "Welcome to a really awesome game!\n";
57          std::cout << "Press Space To Begin or escape to quit!\n";
58      }
59  };
```

main file

To test the automata, we simply need to 'prime' it on line 61 by giving it a new **IntroScreen** to execute. This will then handle creation of the other classes as and when necessary. You'll note that unlike last time, we've got a **while** loop that uses a **Window** parameter. This is to allow the test code to check for the keyboard state; were this automata setup to be integrated into game code 'properly', it'd not be needed, and the machine could just be 'Updated' when necessary.

```
60  void TestPushdownAutomata(Window* w) {
61      PushdownMachine machine(new IntroScreen());
62      while (w->UpdateWindow()) {
63          float dt = w->GetTimer()->GetTimeDeltaSeconds();
64          if (!machine.Update(dt)) {
65              return;
66          }
67      }
68  }
```

main file

To test the machine, we just need to call the **TestPushdownAutomata** function just after the **Window** is created in the **main** function.

Conclusion

Pushdown automata allow us to define some complex, but strict, behaviours to the AI in our game. By being able to store a history of states, there's the ability to give our AI agents a simple ordered memory of what they were trying to do recently. They have most of the benefits of finite state machines, including a clear encapsulation of state code, to make more encapsulated, easy to reuse functionality.

In the example shown in this tutorial we cheat a little and instantiate and delete objects for states. This works fine, but does mean that memory is constantly being allocated and destroyed whenever the automata changes. In a 'real' game, a pushdown automata might instead have a single allocation of memory (of a couple of kilobytes in a size, perhaps), with the 'pushing' and 'popping' becoming the moving of a pointer through this memory allocation, that can then have objects allocated within it very cheaply.

Pushdown automata are useful beyond just AI as well, and are a useful way of encapsulating a large number of choices that game code must make, from which game menu to show, to allowing or disallowing certain actions based on user input.

Further Work

1) In the previous tutorial, a subclass of a **GameObject** was created that had an *Update* method to service a state machine. Try making another new **GameObject** subclass that can instead have a pushdown automata to make decisions.

2) Try adding a *Reset* method to the **PushDownMachine** class so that it can be returned back to its first initial state.