Причёсываем код. Машины состояний

Очень часто необходимо, чтобы объект вёл себя определённым образом в зависимости от разных условий – состояний. К примеру, это может быть всплывающее окно, которое может иметь три состояния – появление, стандартная отрисовка, скрытие или же игровой персонаж, который может спокойно стоять, бегать или стрелять. Для определения подобного рода действий может использоваться код

///////////////////////////////

enum class State

{

Wait,

Dance,

Joke,

Move,

Run,

Attack,

};

void Update(float dt)

{

State state = GetCurrentState();

switch (state)

{

case State::Wait:

// animation idle

break;

case State::Move:

// animation move

// calculate new position

break;

case State::Attack:

// animation attack

// shoot the sheriff

break;

}

// очевидно изменяем состояние

state = State::Attack;

}

///////////////////////////////

Но может и техника, которая была для этого создана – машина состояний.

///////////////////////////////

CharacherFSM char\_fsm;

void Update(float dt)

{

char\_fsm.OnUpdate(dt);

…

char\_fsm.ProcessEvent(walk\_away(std::make\_pair(1, 1)));  
}

///////////////////////////////

Машина состояний уже не раз была описана и реализована разными людьми, но либо мне не хватало функционала, либо вместе с одной машиной нужно тащить целую [библиотеку](http://www.boost.org/doc/libs/1_57_0/libs/msm/doc/HTML/). Поэтому было решено залезть под капот и написать свою реализацию. О приключениях при создании будет рассказано в данной статье.

Часть 1. Теория, постановка задачи, имплементация.

Теоретическое определение машины состояний можно почитать на [Вики](https://en.wikipedia.org/wiki/Finite-state_machine), но нам – программистам – необходимо понять какой интерфейс нужен и как будут использовать сие творение. Данный шаблон проектирование планируется использоваться в системах реального времени, например, в играх. И входной/выходной набор символов, который необходим для полного соответствия математической абстракции будет упущен. Итак, что мы имеем.

1. Конечный набор состояний.
2. Начальное состояние.
3. Таблица переходов.
4. Конечное состояние.

И набор ограничений и пожеланий:

1. В машине может быть только одно активное состояние
2. Количество состояний и переходов должно быть масштабируемо.
3. Максимально простой интерфейс. Желаемый код:

/////////////////

StateMachine char\_fsm;

char\_fsm.AddState(Walk());

char\_fsm.AddState(Dance());

char\_fsm.SetTransitions({\_row<Wait, Dance, idle\_action\_completed>()});

…

char\_fsm.Start<FirstState>();

…

char\_fsm.OnUpdate(.1f);

…

char\_fsm.ProcessEvent(run\_away(position(0, 1)));

...

/////////////////

В главном классе машины –*StateMachine* – хранятся все состояния, одно из которых – текущее. В функции *Update* находится код, который переключает состояния. Есть два возможных подхода в переключении состояний:

1. Переход по триггеру: в каждом вызове Update пробегать по таблице переходов и, если триггер активирован, совершать переключение.
2. Переход по событию: извне машины или в одном из состояний передавать событие, а в Update просматривать произошедшее. Если пришло какое-либо событие, и оно совпадает с одним из переходов – переключаемся.

Коренное отличие – кто будет переключателем. В первом варианте переключать состояния будет машина, соответственно логика переключения будет в одном из имплементаций абстрактного *Transition*. Во втором варианте, события приходят извне и передавать события может кто угодно. Оба подхода имеют свои плюсы и минусы и применяются к разным задачам с переменным успехом. Выбор зависит от личных предпочтений. Мой выбор пал на второй вариант. Подобная модель также реализована в [boost::meta state machine](http://www.boost.org/doc/libs/1_57_0/libs/msm/doc/HTML/ch03s02.html#d0e358). Первый подход описан в AI for games (ISBN-13: 978-0123747310).

Первым делом создаём базовые классы события и состояния

///////////////////////////////////////////

struct Event

{

// do not forget about virtual destructor

virtual ~Event() {}

// method for coping events

virtual std::unique\_ptr<Event> Copy() const { return nullptr; }

};

class BaseState

{

public:

virtual ~BaseState() {}

virtual void OnEnter(const Event&) {}

virtual void OnExit() {}

virtual void OnUpdate(float dt) {}

};

///////////////////////////////////////////

Для того, чтобы не нагружать пользователя класса ненужной имплементацией методов, все методы *BaseState* не являются чистыми виртуальными. Часто достаточно имплементировать только один или два, например, обновления и входа. В *Event* объявлен только один виртуальный деструктор, чтобы создалась виртуальная таблица и мы могли использовать магию полиморфизма и кастов.

Набор членов класса машины состояний по сути отражает требования:

/////////////////////////////////////////////

class StateMachine

{

private:

std::vector<std::unique\_ptr<BaseState>> m\_states;

BaseState\* mp\_current;

BaseState\* mp\_next;

std::vector<Transition> m\_transitions;

...

};

/////////////////////////////////////////////

Что до переходов, то это простая структура с тремя полями, которые являются хэш-коду класса-состояния, из которого будет переход, по какому событию переход, в какое состояние переходить.

/////////////////////////////////////////////

struct Transition

{

size\_t m\_hash\_from;

size\_t m\_hash\_to;

size\_t m\_hash\_event;

};

/////////////////////////////////////////////

Теперь нужно выполнить условие старта – с какого-то состояния нужно начать. Есть два возможных сценария: старт с каким-то событием (возможно, понадобится передать дополнительные данные), старт с нуля. Для старта с определённого типа используется шаблон и поиск в векторе по хэш-коду состояния.

/////////////////////////////////////////////

class StateMachine

{

...

private:

BaseState\* FindState(size\_t i\_hash)

{

auto it = std::find\_if(m\_states.begin(), m\_states.end(), [i\_hash](std::unique\_ptr<BaseState>& ip\_state)

{

return i\_hash == typeid(\*ip\_state).hash\_code();

});

if (it != m\_states.end())

return it->get();

return nullptr;

}

...

public:

template<typename State>

void Start()

{

mp\_next = FindState(typeid(State).hash\_code());

if (mp\_next)

mp\_cashed\_event.reset(new Event());

m\_terminate = mp\_next == nullptr;

}

template<typename State>

void Start(const Event& i\_evt)

{

mp\_next = FindState(typeid(State).hash\_code());

if (mp\_next)

mp\_cashed\_event = std::move(i\_evt.Copy());

m\_terminate = mp\_next == nullptr;

}

};

/////////////////////////////////////////////

Для создания переходов воспользуемся шаблонной функцией

/////////////////////////////////////////////

template <typename StateFrom, typename StateTo, typename Event>

Transition \_row()

{

return{ typeid(StateFrom).hash\_code(), typeid(StateTo).hash\_code(), typeid(Event).hash\_code() };

}

class StateMachine

{

...

public:

void AddTransition(Transition i\_trans)

{

m\_transitions.push\_back(i\_trans);

}

void SetTransitions(std::initializer\_list<Transition> transitions)

{

std::copy(transitions.begin(), transitions.end(), std::back\_inserter(m\_transitions));

}

void AddState(std::unique\_ptr<BaseState> ip\_state)

{

m\_states.emplace\_back(std::move(ip\_state));

}

};

/////////////////////////////////////////////

Вот мы и добрались до главного – обновление и **переключение** состояний! Но сначала о жизни простого состояния:

1. Когда выполняется переключение состояния, то необходимо выполнить вход – OnEnter
2. Далее делаем однообразные, или не очень, действия – OnUpdate
3. Затираем следы и хвосты – OnExit

Естественно, перед входом в новое состояние необходимо выйти из текущего, если оно есть. Но сразу вопрос – какое ожидаемое поведение при событии, которое будет посылаться из OnEnter состояния? По нашей логике – выйти из текущего (то есть того, что послало событие) и зайти в следующее. Проблема в том, что текущее состояние может быть ещё не до конца инициализировано и падение или неожиданное поведение может настичь пользователя в самый неподходящий момент. Конечно, можно жёстко следить за входом/выходом своих состояний, но проще, если подобная ситуация вообще не возникнет. Делаем дополнительное ограничение – переход возможен только в *StateMachine::OnUpdate*, а в ProcessEvent только кэшируем событие и указатель на следующее состояние.

/////////////////////////////////////////////

class StateMachine

{

private:

void CheckNextStateTransition()

{

if (mp\_next == nullptr && !m\_terminate)

return;

if (mp\_current != nullptr)

mp\_current->OnExit();

mp\_current = mp\_next;

if (mp\_cashed\_event)

mp\_current->OnEnter(\*mp\_cashed\_event);

else

{

Event base\_event;

mp\_current->OnEnter(base\_event);

}

mp\_next = nullptr;

mp\_cashed\_event.reset();

}

...

public:

void ProcessEvent(std::unique\_ptr<Event> ip\_event)

{

if (mp\_current == nullptr)

return;

const size\_t cur\_hash = typeid(\*mp\_current).hash\_code();

const size\_t ev\_hash = typeid(\*ip\_event).hash\_code();

for (auto& transition : m\_transitions)

{

if (cur\_hash == transition.m\_hash\_from && ev\_hash == transition.m\_hash\_event)

{

auto p\_next = FindState(transition.m\_hash\_to);

if (p\_next)

{

mp\_next = p\_next;

mp\_cashed\_event = std::move(ip\_event);

}

break;

}

}

}

void OnUpdate(float dt)

{

CheckNextStateTransition();

if (mp\_current)

mp\_current->OnUpdate(dt);

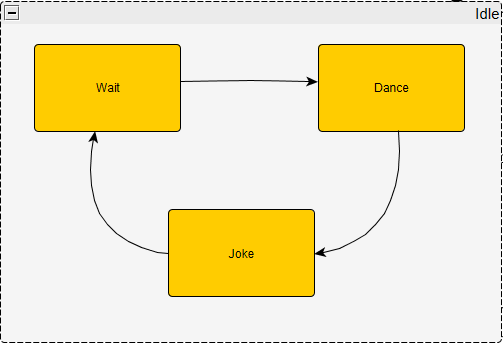
}

...

};

/////////////////////////////////////////////

Что ж, теперь можем попробовать создать тестовую машину состояний



В конструкторе каждого состояния передаём ссылку на родительскую машину, чтобы можно было отсылать события из самого состояния.

/////////////////////////////////////////////

struct idle\_action\_completed : public Event {};

struct Dance : public BaseState

{

StateMachine& m\_parent\_fsm;

Dance(StateMachine& i\_fsm) : m\_parent\_fsm(i\_fsm) {}

virtual void OnUpdate(float dt) override

{

std::cout << "\t[Dance] OnUpdate" << std::endl;

m\_parent\_fsm.ProcessEvent(std::make\_unique<idle\_action\_completed>());

}

};

struct Joke : public BaseState

{

StateMachine& m\_parent\_fsm;

Joke(StateMachine& i\_fsm) : m\_parent\_fsm(i\_fsm) {}

virtual void OnUpdate(float dt) override

{

std::cout << "\t[Joke] OnUpdate" << std::endl;

m\_parent\_fsm.ProcessEvent(std::make\_unique<idle\_action\_completed>());

}

};

struct Wait : public BaseState

{

StateMachine& m\_parent\_fsm;

Wait(StateMachine& i\_fsm) : m\_parent\_fsm(i\_fsm) {}

virtual void OnUpdate(float dt) override

{

std::cout << "\t[Wait] OnUpdate" << std::endl;

m\_parent\_fsm.ProcessEvent(std::make\_unique<idle\_action\_completed>());

}

};

/////////////////////////////////////////////

Так как мы не привязаны ни к каким типам, то для создания машины будет отвечать функция, которая будет конструировать машину. Так как в состояниях используется ссылка на родительскую машину, то возвращать построенную машину по значению нельзя – поменяется адрес, поэтому передаём её как выходной параметр в функцию.

/////////////////////////////////////////////

void IdleFsm(StateMachine& o\_fsm)

{

o\_fsm.AddState(std::make\_unique<Wait>(o\_fsm));

o\_fsm.AddState(std::make\_unique<Dance>(o\_fsm));

o\_fsm.AddState(std::make\_unique<Joke>(o\_fsm));

o\_fsm.SetTransitions({

\_row<Wait, Dance, idle\_action\_completed>(),

\_row<Dance, Joke, idle\_action\_completed>(),

\_row<Joke, Wait, idle\_action\_completed>(),

});

o\_fsm.Start<Wait>();

}

void TestIdle()

{

StateMachine idle\_fsm;

IdleFsm(idle\_fsm);

idle\_fsm.Start<Wait>();

// or we can start with event

// idle\_fsm.Start<Wait>(idle\_action\_completed());

for (int i = 0; i < 10; ++i)

idle\_fsm.OnUpdate(.1f);

}

/////////////////////////////////////////////

***И что написать в выводах?***

Где добавить про компоновочные состояния – машина состояний в машине состояний? Если здесь, то получится слишком нагруженная статья. Возможно, в третьей части, где будет сравнение проводиться – мол, добавочные классы (они достаточно мелкие), но хотелось бы сравнение оставить сравнением, хотя придётся ещё буст туда притягивать (*или его притянуть во второй части, где будет шаблонная реализация*)

Часть 2. А как же шаблоны?

Часть 3. Сравнение