AGENDA

* **INTRO**
  * A Library for Video Games
  * High Level Design Overview

* SIMPLE STATE MACHINES
  * Example - Player Character
  * Library Interface for Simple State Machines
  * Example - Player Character, FSM

* HIERARCHICAL STATE MACHINES
  * Example - Complex Player Character
  * Library Interface for Hierarchical State Machines
  * Example - Complex Player Character, FSM
  * State Machine Complexity Intuition

* OUTRO
  * Advanced Library Interface
  * Post-Mortem
  * Future Work

* SUMMARY

INTRO

* Please, silence your phones

INTRO

* Please, silence your phones
* **Beware of pseudo-code** ☺

INTRO

* Please, silence your phones
* Beware of pseudo-code ☺
* **Raise your hand to ask questions as they come**

DIRTY BOMB VIDEO

INTRO :: Portfolio

* 15 years commercial c++ development

INTRO :: Portfolio

* 15 years commercial c++ development
* **Desktop & embedded development experience early on**

INTRO :: Portfolio

* 15 years commercial c++ development
* Desktop & embedded development experience early on
* **10 years in gamedev as a gameplay / multiplayer / animation coder**

INTRO :: Portfolio

* 15 years commercial c++ development
* Desktop & embedded development experience early on
* 10 years in gamedev as a gameplay / multiplayer / animation coder

INTRO :: History

* Started HFSM in ~2011 in C#/Unity
* By 2011 was (theoretically ☺) convinced that the use of a <u>hierarchical</u> FSM framework in gameplay code should be a HUGE win

INTRO :: History

---

* Started HFSM in ~2011 in C#/Unity
* By 2011 was (theoretically ☺) convinced that the use of a hierarchical FSM framework in gameplay code should be a HUGE win

* **At the time was the only coder on a small (2-4 people) team**
* **Invested a week to design and implement a hierarchical FSM framework**

INTRO :: History

---

* Started HFSM in ~2011 in C#/Unity
* By 2011 was (theoretically ☺) convinced that the use of a hierarchical FSM framework in gameplay code should be a HUGE win

* At the time was the only coder on a small (2-4 people) team
* Invested a week to design and implement a hierarchical FSM framework

* **Threw away 2 prototypes**
* **The 3rd one was usable / stable**

INTRO :: History

* Used it to implement most of the game objects in the game:
  * player character
  * doors
  * UI flow
  * etc.

INTRO :: History

* Used it to implement most of the game objects in the game:
  * player character
  * doors
  * UI flow
  * etc.

* **Using it - was by far the best coding experience:**
  * **fewest bugs**
  * **least time spent on expanding existing objects with new features**

INTRO :: History

---

* Used it to implement most of the game objects in the game:
  * player character
  * doors
  * UI flow
  * etc.

* Using it - was by far the best coding experience:
  * fewest bugs
  * least time spent on expanding existing objects with new features

* **While <u>simple</u> FSMs are great for implementing a <u>sequential</u> feature**
* **<u>Hierarchical</u> FSMs are awesome for combining <u>multiple</u> features within one object**

AGENDA

* INTRO
  * **A Library for Video Games**
  * High Level Design Overview

* SIMPLE STATE MACHINES
  * Example - Player Character
  * Library Interface for Simple State Machines
  * Example - Player Character, FSM

* HIERARCHICAL STATE MACHINES
  * Example - Complex Player Character
  * Library Interface for Hierarchical State Machines
  * Example - Complex Player Character, FSM
  * State Machine Complexity Intuition

* OUTRO
  * Advanced Library Interface
  * Post-Mortem
  * Future Work

* SUMMARY

16.7ms

16.7ms

Time Budget for 1 Frame

16.7ms

Time Budget for 1 Frame

The Most Important Number in GameDev ☺

A Library for Video Games :: Resource Budgets

Business goals ($$) => target hardware specs *(lower requirements mean wider audience)*:

A Library for Video Games :: Resource Budgets

Business goals ($$) => target hardware specs *(lower requirements mean wider audience):*

* **fixed target hardware specs *(minimal + recommended)***

A Library for Video Games :: Resource Budgets

Business goals ($$) => target hardware specs *(lower requirements mean wider audience):*

* fixed target hardware specs *(minimal + recommended)*
* **fixed minimal frame-rate *(e.g. 60 FPS)***

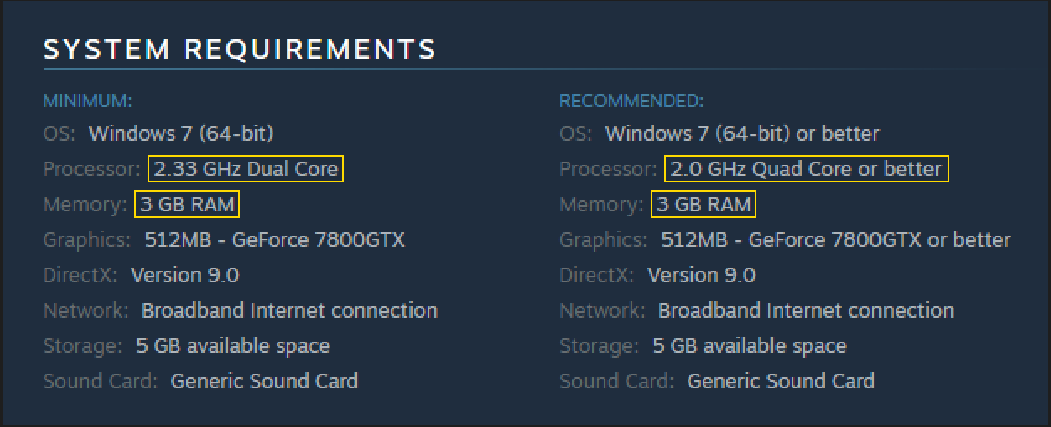
A Library for Video Games :: Resource Budgets

Business goals ($$) => target hardware specs *(lower requirements mean wider audience):*
* fixed target hardware specs *(minimal + recommended)*
* fixed minimal frame-rate *(e.g. 60 FPS)*
  * **soft requirement for desktop / console**
  * **hard requirement for VR**

A Library for Video Games :: Resource Budgets

Business goals ($$) => target hardware specs *(lower requirements mean wider audience):*

* fixed target hardware specs *(minimal + recommended)*
* fixed minimal frame-rate *(e.g. 60 FPS)*
  * soft requirement for desktop / console
  * hard requirement for VR



SYSTEM REQUIREMENTS

MINIMUM:		RECOMMENDED:	
OS:	Windows 7 (64-bit)	OS:	Windows 7 (64-bit) or better
Processor:	2.33 GHz Dual Core	Processor:	2.0 GHz Quad Core or better
Memory:	3 GB RAM	Memory:	3 GB RAM
Graphics:	512MB - GeForce 7800GTX	Graphics:	512MB - GeForce 7800GTX or better
DirectX:	Version 9.0	DirectX:	Version 9.0
Network:	Broadband Internet connection	Network:	Broadband Internet connection
Storage:	5 GB available space	Storage:	5 GB available space
Sound Card:	Generic Sound Card	Sound Card:	Generic Sound Card

A Library for Video Games :: Deterministic Performance

* Memory

A Library for Video Games :: Deterministic Performance

* Memory
  * ~~Dynamic allocations~~ => static / stack allocations
  * ~~Default new()~~ => custom memory management
  * ~~Object-oriented design~~ => cache friendly data layout optimisations

A Library for Video Games :: Deterministic Performance

* Memory
  * ~~Dynamic allocations~~ => static / stack allocations
  * ~~Default new()~~ => custom memory management
  * ~~Object-oriented design~~ => cache friendly data layout optimisations

* **Libraries**

A Library for Video Games :: Deterministic Performance

* Memory
  * ~~Dynamic allocations~~ => static / stack allocations
  * ~~Default new()~~ => custom memory management
  * ~~Object-oriented design~~ => cache friendly data layout optimisations

* Libraries
  * ~~STL~~ => EASTL / custom libraries in most commercial engines
  * ~~std::function~~ => stdext::inplace_function
  * etc.

A Library for Video Games :: .update() Loop

A game engine periodically updates:

A Library for Video Games :: .update() Loop

A game engine periodically updates:

* **subsystems (video, audio, network, etc.)**

A Library for Video Games :: .update() Loop

A game engine periodically updates:
* subsystems (video, audio, network, etc.)
* **game objects (a.k.a. entities)**

A Library for Video Games :: .update() Loop

A game engine periodically updates:

* subsystems (video, audio, network, etc.)
* game objects (a.k.a. entities)
* **at a fixed rate (a.k.a. frame rate)**

A Library for Video Games :: .update() Loop

A game engine periodically updates:
* subsystems (video, audio, network, etc.)
* game objects (a.k.a. entities)
* at a fixed rate (a.k.a. frame rate)

Naturally, a library targeting video games needs to support this.

AGENDA

* INTRO
  * A Library for Video Games
  * **High Level Design Overview**

* SIMPLE STATE MACHINES
  * Example - Player Character
  * Library Interface for Simple State Machines
  * Example - Player Character, FSM

* HIERARCHICAL STATE MACHINES
  * Example - Complex Player Character
  * Library Interface for Hierarchical State Machines
  * Example - Complex Player Character, FSM
  * State Machine Complexity Intuition

* OUTRO
  * Advanced Library Interface
  * Post-Mortem
  * Future Work

* SUMMARY

High Level Design Overview :: Priorities

PRIORITIES: => DECISIONS

High Level Design Overview :: Priorities

PRIORITIES: => DECISIONS

1. Predicable performance -> Fully static structure, built with variadic templates

High Level Design Overview :: Priorities

```
PRIORITIES:                 =>   DECISIONS
1. Predicable performance   ->     Fully static structure, built with variadic templates
2. Safety                   ->     Compile-time error reporting
```

High Level Design Overview :: Priorities

```
PRIORITIES:                    =>   DECISIONS
1. Predicable performance      ->     Fully static structure, built with variadic templates
2. Safety                      ->     Compile-time error reporting
3. Easy to get started         ->     Minimal amount of boilerplate code, base classes, etc.
```

High Level Design Overview :: Priorities

```
PRIORITIES:                   =>   DECISIONS
1. Predicable performance     ->     Fully static structure, built with variadic templates
2. Safety                     ->     Compile-time error reporting
3. Easy to get started        ->     Minimal amount of boilerplate code, base classes, etc.
4. Convenience of use         ->     Delicious sugar sprinkled everywhere ☺
```

High Level Design Overview :: Priorities

```
PRIORITIES:                   =>   DECISIONS
1. Predicable performance     ->     Fully static structure, built with variadic templates
2. Safety                     ->     Compile-time error reporting
3. Easy to get started        ->     Minimal amount of boilerplate code, base classes, etc.
4. Convenience of use         ->     Delicious sugar sprinkled everywhere ☺
5. Rich feature set           ->     Advanced features available for advanced users
```

High Level Design Overview :: 'Proactive' FSM

'PROACTIVE' APPROACH TO FSM DESIGN

High Level Design Overview :: 'Proactive' FSM

'PROACTIVE' APPROACH TO FSM DESIGN

* **State doesn’t leak outside of the FSM! (e.g. no UML state guards)**

High Level Design Overview :: 'Proactive' FSM

'PROACTIVE' APPROACH TO FSM DESIGN
* State doesn’t leak outside of the FSM! (e.g. no UML state guards)
* **Owner’s code shouldn’t care about ‘Which state the FSM is in?’**

High Level Design Overview :: 'Proactive' FSM

'PROACTIVE' APPROACH TO FSM DESIGN

* State doesn’t <u>leak</u> outside of the FSM! (e.g. no UML state guards)
* Owner’s code shouldn’t care about ‘Which state the FSM is in?’
* **Instead, the FSM takes control of the owner’s object (via Context interface)**

High Level Design Overview :: 'Proactive' FSM

'PROACTIVE' APPROACH TO FSM DESIGN
* State doesn’t leak outside of the FSM! (e.g. no UML state guards)
* Owner’s code shouldn’t care about ‘Which state the FSM is in?’
* Instead, the FSM takes control of the owner’s object (via Context interface)

EXAMPLE

High Level Design Overview :: 'Proactive' FSM

'PROACTIVE' APPROACH TO FSM DESIGN
* State doesn’t <u>leak</u> outside of the FSM! (e.g. no UML state guards)
* Owner’s code shouldn’t care about ‘Which state the FSM is in?’
* Instead, the FSM takes control of the owner’s object (via Context interface)

EXAMPLE
* **Think of a ‘brain’ for an AI soldier in a game**

High Level Design Overview :: 'Proactive' FSM

'PROACTIVE' APPROACH TO FSM DESIGN

* State doesn’t <u>leak</u> outside of the FSM! (e.g. no UML state guards)
* Owner’s code shouldn’t care about ‘Which state the FSM is in?’
* Instead, the FSM takes control of the owner’s object (via Context interface)

EXAMPLE

* Think of a ‘brain’ for an AI soldier in a game
* **The owner object has a mesh, animations, sounds, etc.**

High Level Design Overview :: 'Proactive' FSM

'PROACTIVE' APPROACH TO FSM DESIGN
* State doesn’t leak outside of the FSM! (e.g. no UML state guards)
* Owner’s code shouldn’t care about ‘Which state the FSM is in?’
* Instead, the FSM takes control of the owner’s object (via Context interface)

EXAMPLE
* Think of a ‘brain’ for an AI soldier in a game
* The owner object has a mesh, animations, sounds, etc.
* **FSM is in control of it, using all of those to fake a 'living' human**

High Level Design Overview :: 'Proactive' vs 'Reactive' FSM

NO UML-STYLE EVENT REACTIONS

High Level Design Overview :: 'Proactive' vs 'Reactive' FSM

NO UML-STYLE EVENT REACTIONS

* A programmer knows what the target state is for any transition

High Level Design Overview :: 'Proactive' vs 'Reactive' FSM

NO UML-STYLE EVENT REACTIONS

* A programmer knows what the target state is for any transition
* **There's no real need for event → transition indirection (in the general case)**

High Level Design Overview :: 'Proactive' vs 'Reactive' FSM

NO UML-STYLE EVENT REACTIONS

* A programmer knows what the target state is for any transition
* There’s no real need for event → transition indirection (in the general case)
* **If you still want it - event handling is trivial to add that on top of a 'proactive' FSM**

AGENDA

* INTRO
  * A Library for Video Games
  * High Level Design Overview

* SIMPLE STATE MACHINES
  * **Example - Player Character**
  * Library Interface for Simple State Machines
  * Example - Player Character, FSM

* HIERARCHICAL STATE MACHINES
  * Example - Complex Player Character
  * Library Interface for Hierarchical State Machines
  * Example - Complex Player Character, FSM
  * State Machine Complexity Intuition

* OUTRO
  * Advanced Library Interface
  * Post-Mortem
  * Future Work

* SUMMARY

Example - Player Character

Let's start with simple weapon operation sequence: **Idle**

Example - Player Character

Let's start with simple weapon operation sequence: Idle → **Fire**

Example - Player Character

Let's start with simple weapon operation sequence: Idle → Fire → **Reload**

Example - Player Character :: enum/switch Approach

```
struct PlayerCharacter {
                                                     };
```

Example - Player Character :: enum/switch Approach

```
struct PlayerCharacter {
    enum State { Idle, Firing, Reloading };

    State _state;

};
```

Example - Player Character :: enum/switch Approach

```
struct PlayerCharacter {
    enum State { Idle, Firing, Reloading };

    State _state;

    void uberUpdate(const float deltaTime) {

    }
};
```

Example - Player Character :: enum/switch Approach

```
struct PlayerCharacter {
    enum State { Idle, Firing, Reloading };

    State _state;

    void uberUpdate(const float deltaTime) {
        switch (_state) {
        case Idle:
            // ...
            break;
        case Firing:
            // ...
            break;
        case Reloading:
            // ...
            break;
        default:
            // error detection
        }
    }
};
```

Example - Player Character :: enum/switch Approach

```
struct PlayerCharacter {
    enum State { Idle, Firing, Reloading };

    State _state;

    void uberUpdate(const float deltaTime) {
        switch (_state) {
        case Idle:
            // ...
            break;
        case Firing:
            // ...
            break;
        case Reloading:
            // ...
            break;
        default:
            // error detection
        }
    }
};
```

* Fine for something as simple as 3-state sequence
* Handling state transitions might get a bit messier

AGENDA

* INTRO
  * A Library for Video Games
  * High Level Design Overview

* SIMPLE STATE MACHINES
  * Example - Player Character
  * **Library Interface for Simple State Machines**
  * Example - Player Character, FSM

* HIERARCHICAL STATE MACHINES
  * Example - Complex Player Character
  * Library Interface for Hierarchical State Machines
  * Example - Complex Player Character, FSM
  * State Machine Complexity Intuition

* OUTRO
  * Advanced Library Interface
  * Post-Mortem
  * Future Work

* SUMMARY

Library Interface for Simple State Machines :: Integration Boilerplate

To use HFSM in your project:

```
// 0: download it at:
//    https://github.com/andrew-gresyk/HFSM.git
```

Library Interface for Simple State Machines :: Integration Boilerplate

To use HFSM in your project:

```
// 0: download it at:
//    https://github.com/andrew-gresyk/HFSM.git

// 1: include HFSM header
#include <hfsm.h>
```

Library Interface for Simple State Machines :: Integration Boilerplate

To use HFSM in your project:

```
// 0: download it at:
//    https://github.com/andrew-gresyk/HFSM.git

// 1: include HFSM header
#include <hfsm.h>

// 2: define interface between the FSM and its owner
//    also ok to use the owner object itself
struct Context { /* ... */ };
```

Library Interface for Simple State Machines :: Integration Boilerplate

To use HFSM in your project:

```
// 0: download it at:
//    https://github.com/andrew-gresyk/HFSM.git

// 1: include HFSM header
#include <hfsm.h>

// 2: define interface between the FSM and its owner
//    also ok to use the owner object itself
struct Context { /* ... */ };

// 3: [optional] typedef FSM class for convenience
using M = hfsm::Machine<Context>;
```

Library Interface for Simple State Machines :: Integration Boilerplate

To use HFSM in your project:

```
// 0: download it at:
//    https://github.com/andrew-gresyk/HFSM.git

// 1: include HFSM header
#include <hfsm.h>

// 2: define interface between the FSM and its owner
//    also ok to use the owner object itself
struct Context { /* ... */ };

// 3: [optional] typedef FSM class for convenience
using M = hfsm::Machine<Context>;

// 4: define states
struct Idle : M::Base { /* ... */ };
struct Firing : M::Timed { /* ... */ };
struct Reloading : M::Timed { /* ... */ };
```

Library Interface for Simple State Machines :: Integration Boilerplate

To use HFSM in your project:

```
// 0: download it at:
//    https://github.com/andrew-gresyk/HFSM.git

// 1: include HFSM header
#include <hfsm.h>

// 2: define interface between the FSM and its owner
//    also ok to use the owner object itself
struct Context { /* ... */ };

// 3: [optional] typedef FSM class for convenience
using M = hfsm::Machine<Context>;

// 4: define states
struct Idle : M::Base { /* ... */ };
struct Firing : M::Timed { /* ... */ };
struct Reloading : M::Timed { /* ... */ };
```

```
// 5: define FSM structure
using PlayerFSM = M::CompositeRoot<
    M::State<Idle>,
    M::State<Firing>,
    M::State<Reloading>
>;
```

Library Interface for Simple State Machines :: Integration Boilerplate

To use HFSM in your project:

```cpp
// 0: download it at:
//    https://github.com/andrew-gresyk/HFSM.git

// 1: include HFSM header
#include <hfsm.h>

// 2: define interface between the FSM and its owner
//    also ok to use the owner object itself
struct Context { /* ... */ };

// 3: [optional] typedef FSM class for convenience
using M = hfsm::Machine<Context>;

// 4: define states
struct Idle : M::Base { /* ... */ };
struct Firing : M::Timed { /* ... */ };
struct Reloading : M::Timed { /* ... */ };
```

```cpp
// 5: define FSM structure
using PlayerFSM = M::CompositeRoot<
    M::State<Idle>,
    M::State<Firing>,
    M::State<Reloading>
>;

// 6: create FSM instance
void start() {
    Context c;
    PlayerFSM fsm(c);

    fsm.enter();
}
```

Library Interface for Simple State Machines :: Integration Boilerplate

To use HFSM in your project:

```cpp
// 0: download it at:
//    https://github.com/andrew-gresyk/HFSM.git

// 1: include HFSM header
#include <hfsm.h>

// 2: define interface between the FSM and its owner
//    also ok to use the owner object itself
struct Context { /* ... */ };

// 3: [optional] typedef FSM class for convenience
using M = hfsm::Machine<Context>;

// 4: define states
struct Idle : M::Base { /* ... */ };
struct Firing : M::Timed { /* ... */ };
struct Reloading : M::Timed { /* ... */ };
```

```cpp
// 5: define FSM structure
using PlayerFSM = M::CompositeRoot<
    M::State<Idle>,
    M::State<Firing>,
    M::State<Reloading>
>;

// 6: create FSM instance
void start() {
    Context c;
    PlayerFSM fsm(c);

    fsm.enter();
}

// 7: set up periodic updates
void update(const float detlaTime) {
    fsm.update(detlaTime);
}
```

Library Interface for Simple State Machines :: Anatomy of a State

HFSM uses static polymorphism, no need to make methods virtual:

```
struct Idle
    : M::Base // sugar, adds M::Control, M::Context, M::etc. into local scope
{









};
```

Library Interface for Simple State Machines :: Anatomy of a State

HFSM uses static polymorphism, no need to make methods virtual:

```
struct Idle
    : M::Base // sugar, adds M::Control, M::Context, M::etc. into local scope
{




    // a.k.a. begin() / ctor / etc.
    void enter(Context& context, const Time time);








    // a.k.a. end() / dtor / etc.
    void leave(Context& context, const Time time);
};
```

Library Interface for Simple State Machines :: Anatomy of a State

HFSM uses static polymorphism, no need to make methods virtual:

```
struct Idle
    : M::Base // sugar, adds M::Control, M::Context, M::etc. into local scope
{



    // a.k.a. begin() / ctor / etc.
    void enter(Context& context, const Time time);

    // called on recursively on all active states once per frame
    void update(Context& context, const Time time);




    // a.k.a. end() / dtor / etc.
    void leave(Context& context, const Time time);
};
```

Library Interface for Simple State Machines :: Anatomy of a State

HFSM uses static polymorphism, no need to make methods virtual:

```
struct Idle
    : M::Base // sugar, adds M::Control, M::Context, M::etc. into local scope
{



    // a.k.a. begin() / ctor / etc.
    void enter(Context& context, const Time time);

    // called on recursively on all active states once per frame
    void update(Context& context, const Time time);

    // localised place for the state to request transitions
    void transition(Control& control, Context& context, const Time time);

    // a.k.a. end() / dtor / etc.
    void leave(Context& context, const Time time);
};
```

Library Interface for Simple State Machines :: Anatomy of a State

---

HFSM uses static polymorphism, no need to make methods virtual:

```
struct Idle
    : M::Base // sugar, adds M::Control, M::Context, M::etc. into local scope
{
    // serves the same purpose as UML's "guard condition"
    void substitute(Control& control, Context& context, const Time time);

    // a.k.a. begin() / ctor / etc.
    void enter(Context& context, const Time time);

    // called on recursively on all active states once per frame
    void update(Context& context, const Time time);

    // localised place for the state to request transitions
    void transition(Control& control, Context& context, const Time time);

    // a.k.a. end() / dtor / etc.
    void leave(Context& context, const Time time);
};
```

Library Interface for Simple State Machines :: Transitions

```
// 1. Initiated from within a state machine, by a state:
struct Idle : M::Base {



}

struct Firing : M::Base { /* .. */ }        // target state
```

Library Interface for Simple State Machines :: Transitions

```
// 1. Initiated from within a state machine, by a state:
struct Idle : M::Base {
    void Idle::transition(Control& control, Context& context, const Time time) {

    }
}

struct Firing : M::Base { /* .. */ }        // target state
```

Library Interface for Simple State Machines :: Transitions

```
// 1. Initiated from within a state machine, by a state:
struct Idle : M::Base {
    void Idle::transition(Control& control, Context& context, const Time time) {
        control.changeTo<Firing>();
    }
}

struct Firing : M::Base { /* .. */ }        // target state
```

Library Interface for Simple State Machines :: Transitions

```cpp
// 1. Initiated from within a state machine, by a state:
struct Idle : M::Base {
    void Idle::transition(Control& control, Context& context, const Time time) {
        control.changeTo<Firing>();
    }
}

struct Firing : M::Base { /* .. */ }        // target state
```

```cpp
// 2. Initiated from the outside of a state machine, using matching functions:
void main() {





}
```

Library Interface for Simple State Machines :: Transitions

```cpp
// 1. Initiated from within a state machine, by a state:
struct Idle : M::Base {
    void Idle::transition(Control& control, Context& context, const Time time) {
        control.changeTo<Firing>();
    }
}

struct Firing : M::Base { /* .. */ }        // target state
```

```cpp
// 2. Initiated from the outside of a state machine, using matching functions:
void main() {
    Context context;
    PlayerFSM fsm(context);

    fsm.enter();



}
```

Library Interface for Simple State Machines :: Transitions

```cpp
// 1. Initiated from within a state machine, by a state:
struct Idle : M::Base {
    void Idle::transition(Control& control, Context& context, const Time time) {
        control.changeTo<Firing>();
    }
}

struct Firing : M::Base { /* .. */ }        // target state
```

```cpp
// 2. Initiated from the outside of a state machine, using matching functions:
void main() {
    Context context;
    PlayerFSM fsm(context);

    fsm.enter();

    fsm.changeTo<Firing>(); // not processed until the following .update()

}
```

Library Interface for Simple State Machines :: Transitions

```
// 1. Initiated from within a state machine, by a state:
struct Idle : M::Base {
    void Idle::transition(Control& control, Context& context, const Time time) {
        control.changeTo<Firing>();
    }
}

struct Firing : M::Base { /* .. */ }        // target state
```

```
// 2. Initiated from the outside of a state machine, using matching functions:
void main() {
    Context context;
    PlayerFSM fsm(context);

    fsm.enter();

    fsm.changeTo<Firing>(); // not processed until the following .update()
    fsm.update(time);       // <-
}
```

Library Interface for Simple State Machines :: State Method Call Sequence

```
template <...>
class Machine {
    void Root::update() {
        Control control;




















    }
};
```

```
/*
iteration 1:
















iteration 2:






*/
```

Library Interface for Simple State Machines :: State Method Call Sequence

```
template <...>
class Machine {
    void Root::update() {
        Control control;

        activeState.update();
        activeState.transition(control);







    }
};
```

```
/*
iteration 1:

    activeState.update();
    activeState.transition() {
        fsm.changeTo<Idle>();
    }






iteration 2:



*/
```

Library Interface for Simple State Machines :: State Method Call Sequence

```
template <...>
class Machine {
    void Root::update() {
        Control control;

        activeState.update();
        activeState.transition(control);

        while (control.requests.size() > 0) {



        }





    }
};
```

```
/*
iteration 1:

    activeState.update();
    activeState.transition() {
        fsm.changeTo<Idle>();
    }






iteration 2:




*/
```

Library Interface for Simple State Machines :: State Method Call Sequence

```
template <...>
class Machine {
    void Root::update() {
        Control control;

        activeState.update();
        activeState.transition(control);

        while (control.requests.size() > 0) {
            nextState = control.requests[0].state;

            nextState.substitute(control);
        }



    }
};
```

```
/*
iteration 1:

    activeState.update();
    activeState.transition() {
        fsm.changeTo<Idle>();
    }

    nextState.substitute() {}




iteration 2:



*/
```

Library Interface for Simple State Machines :: State Method Call Sequence

```cpp
template <...>
class Machine {
    void Root::update() {
        Control control;

        activeState.update();
        activeState.transition(control);

        while (control.requests.size() > 0) {
            nextState = control.requests[0].state;

            nextState.substitute(control);
        }

        if (nextState != activeState) {


        }
    }
};
```

```cpp
/*
iteration 1:

    activeState.update();
    activeState.transition() {
        fsm.changeTo<Idle>();
    }

    nextState.substitute() {}




iteration 2:




*/
```

Library Interface for Simple State Machines :: State Method Call Sequence

```
template <...>
class Machine {
    void Root::update() {
        Control control;

        activeState.update();
        activeState.transition(control);

        while (control.requests.size() > 0) {
            nextState = control.requests[0].state;

            nextState.substitute(control);
        }

        if (nextState != activeState) {
            activeState.leave();
            nextState.enter();
        }
    }
};
```

```
/*
iteration 1:

    activeState.update();
    activeState.transition() {
        fsm.changeTo<Idle>();
    }

    nextState.substitute() {}

    activeState.leave();
    nextState.enter();

iteration 2:



*/
```

Library Interface for Simple State Machines :: State Method Call Sequence

```
template <...>
class Machine {
    void Root::update() {
        Control control;

        activeState.update();
        activeState.transition(control);

        while (control.requests.size() > 0) {
            nextState = control.requests[0].state;

            nextState.substitute(control);
        }

        if (nextState != activeState) {
            activeState.leave();
            nextState.enter();
        }
    }
};
```

```
/*
iteration 1:

    activeState.update();
    activeState.transition() {
        fsm.changeTo<Idle>();
    }

    nextState.substitute() {}

    activeState.leave();
    nextState.enter();

iteration 2:

    nextState.update();
    nextState.transition();
*/
```

AGENDA

* INTRO
  * A Library for Video Games
  * High Level Design Overview

* SIMPLE STATE MACHINES
  * Example - Player Character
  * Library Interface for Simple State Machines
  * **Example - Player Character, FSM**

* HIERARCHICAL STATE MACHINES
  * Example - Complex Player Character
  * Library Interface for Hierarchical State Machines
  * Example - Complex Player Character, FSM
  * State Machine Complexity Intuition

* OUTRO
  * Advanced Library Interface
  * Post-Mortem
  * Future Work

* SUMMARY

Example - Player Character, FSM :: State Diagram

```
Root                // Implicit composite region
 ├ Idle             // state
 ├ Firing           // state
 └ Reloading        // state
```

Example - Player Character, FSM :: C++ Implementation

```
struct Context {
    unsigned weaponAmmoCount;
    unsigned weaponAmmoCapacity;
    unsigned spareAmmoCount;
}
```

```
struct Idle : M::Base {











};
```

Example - Player Character, FSM :: C++ Implementation

```
struct Context {
    unsigned weaponAmmoCount;
    unsigned weaponAmmoCapacity;
    unsigned spareAmmoCount;
}
```

```
struct Idle : M::Base {
    void transition(Control& c, Context& _, const Time t) {







    }




};
```

Example - Player Character, FSM :: C++ Implementation

```
struct Context {
    unsigned weaponAmmoCount;
    unsigned weaponAmmoCapacity;
    unsigned spareAmmoCount;
}
```

```
struct Idle : M::Base {
    void transition(Control& c, Context& _, const Time t) {
        if (_.weaponAmmoCount > 0 && keyPressed(KeyFire))
            c.changeTo<Firing>();







    }





};
```

Example - Player Character, FSM :: C++ Implementation

```
struct Context {
    unsigned weaponAmmoCount;
    unsigned weaponAmmoCapacity;
    unsigned spareAmmoCount;
}
```

```
struct Idle : M::Base {
    void transition(Control& c, Context& _, const Time t) {
        if (_.weaponAmmoCount > 0 && keyPressed(KeyFire))
            c.changeTo<Firing>();
        else if (_.weaponAmmoCount < _.weaponAmmoCapacity &&
                 _.spareAmmoCount > 0 &&
                 (keyPressed(KeyReload) || _.weaponAmmoCount == 0))
        {
            c.changeTo<Reloading>();
        }
    }



};
```

Example - Player Character, FSM :: C++ Implementation

```cpp
struct Context {
    unsigned weaponAmmoCount;
    unsigned weaponAmmoCapacity;
    unsigned spareAmmoCount;
}
```

```cpp
struct Idle : M::Base {
    void transition(Control& c, Context& _, const Time t) {
        if (_.weaponAmmoCount > 0 && keyPressed(KeyFire))
            c.changeTo<Firing>();
        else if (_.weaponAmmoCount < _.weaponAmmoCapacity &&
                 _.spareAmmoCount > 0 &&
                 (keyPressed(KeyReload) || _.weaponAmmoCount == 0))
        {
            c.changeTo<Reloading>();
        }
    }

    void update(Context& c, const Time t) {
        processMovement(t);
    }
};
```

Example - Player Character, FSM :: C++ Implementation

```
struct Context {
    unsigned weaponAmmoCount;
    unsigned weaponAmmoCapacity;
    unsigned spareAmmoCount;
}
```

```
struct Firing : M::Timed {













};
```

Example - Player Character, FSM :: C++ Implementation

```
struct Context {
    unsigned weaponAmmoCount;
    unsigned weaponAmmoCapacity;
    unsigned spareAmmoCount;
}
```

```
struct Firing : M::Timed {
    void enter(Context& _, const Time t) {



    }







};
```

Example - Player Character, FSM :: C++ Implementation

```cpp
struct Context {
    unsigned weaponAmmoCount;
    unsigned weaponAmmoCapacity;
    unsigned spareAmmoCount;
}
```

```cpp
struct Firing : M::Timed {
    void enter(Context& _, const Time t) {
        assert(_.weaponAmmoCount > 0)
        --_.weaponAmmoCount;


    }







};
```

Example - Player Character, FSM :: C++ Implementation

```cpp
struct Context {
    unsigned weaponAmmoCount;
    unsigned weaponAmmoCapacity;
    unsigned spareAmmoCount;
}
```

```cpp
struct Firing : M::Timed {
    void enter(Context& _, const Time t) {
        assert(_.weaponAmmoCount > 0)
        --_.weaponAmmoCount;
        playFiringAnimation();
        dealDamage();
    }








};
```

Example - Player Character, FSM :: C++ Implementation

```cpp
struct Context {
    unsigned weaponAmmoCount;
    unsigned weaponAmmoCapacity;
    unsigned spareAmmoCount;
}
```

```cpp
struct Firing : M::Timed {
    void enter(Context& _, const Time t) {
        assert(_.weaponAmmoCount > 0)
        --_.weaponAmmoCount;
        playFiringAnimation();
        dealDamage();
    }

    void transition(Control& c, Context& _, const Time t) {




    }
};
```

Example - Player Character, FSM :: C++ Implementation

```cpp
struct Context {
    unsigned weaponAmmoCount;
    unsigned weaponAmmoCapacity;
    unsigned spareAmmoCount;
}
```

```cpp
struct Firing : M::Timed {
    void enter(Context& _, const Time t) {
        assert(_.weaponAmmoCount > 0)
        --_.weaponAmmoCount;
        playFiringAnimation();
        dealDamage();
    }

    void transition(Control& c, Context& _, const Time t) {
        if (!keyPressed(KeyFire))
            c.changeTo<Idle>();


    }
};
```

Example - Player Character, FSM :: C++ Implementation

```cpp
struct Context {
    unsigned weaponAmmoCount;
    unsigned weaponAmmoCapacity;
    unsigned spareAmmoCount;
}
```

```cpp
struct Firing : M::Timed {
    void enter(Context& _, const Time t) {
        assert(_.weaponAmmoCount > 0)
        --_.weaponAmmoCount;
        playFiringAnimation();
        dealDamage();
    }

    void transition(Control& c, Context& _, const Time t) {
        if (!keyPressed(KeyFire))
            c.changeTo<Idle>();
        else if (M::Timed::duration() > 1s)
            c.changeTo<Firing>();
    }
};
```

Example - Player Character, FSM :: C++ Implementation

```
struct Context {
    unsigned weaponAmmoCount;
    unsigned weaponAmmoCapacity;
    unsigned spareAmmoCount;
}
```

```
struct Reloading : M::Timed {

};
```

Example - Player Character, FSM :: C++ Implementation

```
struct Context {
    unsigned weaponAmmoCount;
    unsigned weaponAmmoCapacity;
    unsigned spareAmmoCount;
}
```

```
struct Reloading : M::Timed {
    void enter(Control& c, Context& _, const Time t) {

    }








};
```

Example - Player Character, FSM :: C++ Implementation

```
struct Context {
    unsigned weaponAmmoCount;
    unsigned weaponAmmoCapacity;
    unsigned spareAmmoCount;
}
```

```
struct Reloading : M::Timed {
    void enter(Control& c, Context& _, const Time t) {
        assert(_.spareAmmoCount > 0)
    }
};
```

Example - Player Character, FSM :: C++ Implementation

```cpp
struct Context {
    unsigned weaponAmmoCount;
    unsigned weaponAmmoCapacity;
    unsigned spareAmmoCount;
}
```

```cpp
struct Reloading : M::Timed {
    void enter(Control& c, Context& _, const Time t) {
        assert(_.spareAmmoCount > 0)
    }

    void transition(Control& c, Context& _, const Time t) {


    }




};
```

Example - Player Character, FSM :: C++ Implementation

```cpp
struct Context {
    unsigned weaponAmmoCount;
    unsigned weaponAmmoCapacity;
    unsigned spareAmmoCount;
}
```

```cpp
struct Reloading : M::Timed {
    void enter(Control& c, Context& _, const Time t) {
        assert(_.spareAmmoCount > 0)
    }

    void transition(Control& c, Context& _, const Time t) {
        if (M::Timed::duration() > 2s)
            c.changeTo<Idle>();
    }




};
```

Example - Player Character, FSM :: C++ Implementation

```cpp
struct Context {
    unsigned weaponAmmoCount;
    unsigned weaponAmmoCapacity;
    unsigned spareAmmoCount;
}
```

```cpp
struct Reloading : M::Timed {
    void enter(Control& c, Context& _, const Time t) {
        assert(_.spareAmmoCount > 0)
    }

    void transition(Control& c, Context& _, const Time t) {
        if (M::Timed::duration() > 2s)
            c.changeTo<Idle>();
    }

    void leave(Context& _, const Time t) {




    }
};
```

Example - Player Character, FSM :: C++ Implementation

```
struct Context {
    unsigned weaponAmmoCount;
    unsigned weaponAmmoCapacity;
    unsigned spareAmmoCount;
}
```

```
struct Reloading : M::Timed {
    void enter(Control& c, Context& _, const Time t) {
        assert(_.spareAmmoCount > 0)
    }

    void transition(Control& c, Context& _, const Time t) {
        if (M::Timed::duration() > 2s)
            c.changeTo<Idle>();
    }

    void leave(Context& _, const Time t) {
        const unsigned ammoToLoad = std::min(_.spareAmmoCount,
                                    _.weaponAmmoCapacity - _.weaponAmmoCount);


    }
};
```

Example - Player Character, FSM :: C++ Implementation

```
struct Context {
    unsigned weaponAmmoCount;
    unsigned weaponAmmoCapacity;
    unsigned spareAmmoCount;
}
```

```
struct Reloading : M::Timed {
    void enter(Control& c, Context& _, const Time t) {
        assert(_.spareAmmoCount > 0)
    }

    void transition(Control& c, Context& _, const Time t) {
        if (M::Timed::duration() > 2s)
            c.changeTo<Idle>();
    }

    void leave(Context& _, const Time t) {
        const unsigned ammoToLoad = std::min(_.spareAmmoCount,
                                             _.weaponAmmoCapacity - _.weaponAmmoCount);
        _.spareAmmoCount -= ammoToLoad;
        _.weaponAmmoCount += ammoToLoad;
    }
};
```

AGENDA

* INTRO
  * A Library for Video Games
  * High Level Design Overview

* SIMPLE STATE MACHINES
  * Example - Player Character
  * Library Interface for Simple State Machines
  * Example - Player Character, FSM

* HIERARCHICAL STATE MACHINES
  * **Example - Complex Player Character**
  * Library Interface for Hierarchical State Machines
  * Example - Complex Player Character, FSM
  * State Machine Complexity Intuition

* OUTRO
  * Advanced Library Interface
  * Post-Mortem
  * Future Work

* SUMMARY

Example - Complex Player Character :: Object State Matrix

Let’s add ‘Sprinting’ mechanic to the PlayerCharacter
* tactical option, move faster but can’t shoot
* allow weapon reloading if the user has required perk

Example - Complex Player Character :: Object State Matrix

Valid vs. invalid states:

	Idle	Firing	Reloading
Walking	✓	✓	✓
Sprinting	✓	×	✓

Standing = Walking at 0 speed, with a good animator and animation framework ☺

Example - Complex Player Character :: enum/switch Approach

```
struct PlayerCharacter {
    enum { Idle, Firing, Reloading } _state;
    bool _isSprinting;

};
```

Example - Complex Player Character :: enum/switch Approach

```
struct PlayerCharacter {
    enum { Idle, Firing, Reloading } _state;
    bool _isSprinting;

    void uberUpdate(const float detlaTime) {








                                                                    }
                                                                };
```

Example - Complex Player Character :: enum/switch Approach

```
struct PlayerCharacter {
    enum { Idle, Firing, Reloading } _state;
    bool _isSprinting;

    void uberUpdate(const float detlaTime) {
        if (_isSprinting) {

        } else {
        }
    }
};
```

Example - Complex Player Character :: enum/switch Approach

```
struct PlayerCharacter {
    enum { Idle, Firing, Reloading } _state;
    bool _isSprinting;

    void uberUpdate(const float detlaTime) {
        if (_isSprinting) {
            // ...
            switch (_state) {
                case Idle:
                    // ...
                case Firing:
                    // ...
                case Reloading:
                    // ...
                default:
                    // error correction
            }
        } else {                                                          }
                                                                      }
                                                                  };
```

Example - Complex Player Character :: enum/switch Approach

```
struct PlayerCharacter {
    enum { Idle, Firing, Reloading } _state;
    bool _isSprinting;

    void uberUpdate(const float detlaTime) {
        if (_isSprinting) {
            // ...
            switch (_state) {
                case Idle:
                    // ...
                case Firing:
                    // ...
                case Reloading:
                    // ...
                default:
                    // error correction
            }
        } else {
            // ...
            switch (_state) {
                case Idle:
                    // ...
                case Firing:
                    // invalid state recovery
                case Reloading:
                    // ...
                default:
                    // error detection
            }
        }
    }
};
```

Example - Complex Player Character :: enum/switch Approach

```
struct PlayerCharacter {
    enum { Idle, Firing, Reloading } _state;
    bool _isSprinting;

    void uberUpdate(const float detlaTime) {
        if (_isSprinting) {
            // ...
            switch (_state) {
                case Idle:
                    // ...
                case Firing:
                    // ...
                case Reloading:
                    // ...
                default:
                    // error correction
            }
        } else {
            // ...
            switch (_state) {
                case Idle:
                    // ...
                case Firing:
                    // invalid state recovery
                case Reloading:
                    // ...
                default:
                    // error detection
            }
            if (_state) {
                case Idle:
                    // ...
                case Reloading:
                    // ...
                default:
                    // error detection
            }
        }
    }
};
```

Example - Complex Player Character :: enum/switch Approach

```
struct PlayerCharacter {
    enum { Idle, Firing, Reloading } _state;
    bool _isSprinting;

    void uberUpdate(const float detlaTime) {
        if (_isSprinting) {
            // ...
            switch (_state) {
                case Idle:
                    // ...
                case Firing:
                    // ...
                case Reloading:
                    // ...
                default:
                    // error correction
            }
        } else {
            // ...
            switch (_state) {
                case Idle:
                    // ...
                case Firing:
                    // invalid state recovery
                case Reloading:
                    // ...
                default:
                    // error detection
            }
            if (_state) {
                case Idle:
                    // ...
                case Reloading:
                    // ...
                default:
                    // error detection
            }
        }
    }
};
```

With every feature added, complexity tends to grow disproportionately :(
To an extent this is also true for a mixed FSM + plain state variable approach

Example - Complex Player Character :: Actual AA/AAA Project Statistics for Better Perspective

Average size of "PlayerCharacter.cpp":

* 10k - 15k+ LOC

Example - Complex Player Character :: Actual AA/AAA Project Statistics for Better Perspective

Average size of "PlayerCharacter.cpp":
* 10k - 15k+ LOC

Average size of "PlayerCharacter::update()":
* **1.5k - 2.0k+**

Example - Complex Player Character :: Actual AA/AAA Project Statistics for Better Perspective

Average size of "PlayerCharacter.cpp":
* 10k - 15k+ LOC

Average size of "PlayerCharacter::update()":
* 1.5k - 2.0k+

Max number of repeating conditional expressions in PlayerCharacter::update():
* **5!!!**

Example - Complex Player Character :: Actual AA/AAA Project Statistics for Better Perspective

Average size of "PlayerCharacter.cpp":
* 10k - 15k+ LOC

Average size of "PlayerCharacter::update()":
* 1.5k - 2.0k+

Max number of repeating conditional expressions in PlayerCharacter::update():
* 5!!!

Total number of different state machine implementations:
* **20+, from very simple to rather complex ones, with state guards, state inheritance, etc.**

AGENDA

* INTRO
  * A Library for Video Games
  * High Level Design Overview

* SIMPLE STATE MACHINES
  * Example - Player Character
  * Library Interface for Simple State Machines
  * Example - Player Character, FSM

* HIERARCHICAL STATE MACHINES
  * Example - Complex Player Character
  * **Library Interface for Hierarchical State Machines**
  * Example - Complex Player Character, FSM
  * State Machine Complexity Intuition

* OUTRO
  * Advanced Library Interface
  * Post-Mortem
  * Future Work

* SUMMARY

Library Interface for Hierarchical State Machines :: Design Approach

Adding hierarchy to an FSM brings up many questions:

Library Interface for Hierarchical State Machines :: Design Approach

Adding hierarchy to an FSM brings up many questions:
* **What is an 'active state' now?**

Library Interface for Hierarchical State Machines :: Design Approach

Adding hierarchy to an FSM brings up many questions:
* What is an 'active state' now?
* **How state transitions should work**?

Library Interface for Hierarchical State Machines :: Design Approach

Adding hierarchy to an FSM brings up many questions:
* What is an 'active state' now?
* How state transitions should work?

No single 'right' answer

Library Interface for Hierarchical State Machines :: Design Approach

Adding hierarchy to an FSM brings up many questions:
* What is an 'active state' now?
* How state transitions should work?

No single 'right' answer

Which is fine, so long as:

Library Interface for Hierarchical State Machines :: Design Approach

Adding hierarchy to an FSM brings up many questions:
* What is an 'active state' now?
* How state transitions should work?

No single 'right' answer

Which is fine, so long as:
* **one rule set is defined**

Library Interface for Hierarchical State Machines :: Design Approach

Adding hierarchy to an FSM brings up many questions:
* What is an 'active state' now?
* How state transitions should work?

No single 'right' answer

Which is fine, so long as:
* one rule set is defined
* **both framework itself and code using the framework adhere to it**

Library Interface for Hierarchical State Machines :: Design Approach

Adding hierarchy to a FSM brings up many questions:
* What is an 'active state' now?
* How state transitions should work?

No single 'right' answer

Which is fine, so long as
* one rule set is defined
* both framework itself and code using the framework adhere to it

So let's define a practical one!

Notation:

Notation:

Root

Notation:

```
Root
 └ State                    // leaf state
```

Library Interface for Hierarchical State Machines :: Building Blocks

Notation:

```
Root
 ├ State                // leaf state
 └ CompositeRegion      // only one active sub-state
```

Library Interface for Hierarchical State Machines :: Building Blocks

Notation:

```
Root
 ├ State                  // leaf state
 └ CompositeRegion        // only one active sub-state
    ├ State
    └ State
```

Notation:

```
Root
 ├  State                   // leaf state
 ├  CompositeRegion         // only one active sub-state
 │   ├  State
 │   └  State
 └  OrthogonalRegion        // all sub-states are active
```

Library Interface for Hierarchical State Machines :: Building Blocks

Notation:

```
Root
  ├ State                    // leaf state
  ├ CompositeRegion          // only one active sub-state
  │ ├ State
  │ └ State
  └ OrthogonalRegion         // all sub-states are active
    ╟ CompositeRegion
    ║
    ║
    ╙ State
```

Library Interface for Hierarchical State Machines :: Building Blocks

Notation:

```
 Root
  ├ State                 // leaf state
  ├ CompositeRegion       // only one active sub-state
  │ ├ State
  │ └ State
  └ OrthogonalRegion      // all sub-states are active
    ╟ CompositeRegion
    ║ ├ State
    ║ └ State
    ╙ State
```

Library Interface for Hierarchical State Machines :: Active Chain

Single active state in a simple FSM becomes an active chain in a hierarchical one:

Library Interface for Hierarchical State Machines :: Active Chain

Single active state in a simple FSM becomes an active chain in a hierarchical one:

1. **Only one active chain exists at any point in time**

Library Interface for Hierarchical State Machines :: Active Chain

Single active state in a simple FSM becomes an active chain in a hierarchical one:

1. Only one active chain exists at any point in time
2. **Starts at a root, ends at one ore more leaf nodes**

Library Interface for Hierarchical State Machines :: Active Chain

Single active state in a simple FSM becomes an active chain in a hierarchical one:

1. Only one active chain exists at any point in time
2. Starts at a root, ends at one ore more leaf nodes
3. **Transitioning to any state in a chain activates the entire chain**

Whenever a state / region is activated:

Library Interface for Hierarchical State Machines :: Transitions Rules

Whenever a state / region is activated:

1. **All parents of the newly activated state also become active**

Library Interface for Hierarchical State Machines :: Transitions Rules

Whenever a state / region is activated:

1. All parents of the newly activated state also become active
2. **For an active composite region - initial sub-state is activated**

Library Interface for Hierarchical State Machines :: Transitions Rules

Whenever a state / region is activated:

1. All parents of the newly activated state also become active
2. For an active composite region - initial sub-state is activated
3. **For an active orthogonal region - all sub-states get activated**

Library Interface for Hierarchical State Machines :: Transitions Rules

---

Whenever a state / region is activated:

1. All parents of the newly activated state also become active
2. For an active composite region - initial sub-state is activated
3. For an active orthogonal region - all sub-states get activated

---

Example:

Library Interface for Hierarchical State Machines :: Transitions Rules

---

Whenever a state / region is activated:
1. All parents of the newly activated state also become active
2. For an active composite region - initial sub-state is activated
3. For an active orthogonal region - all sub-states get activated

---

Example:

```
Root
 ├ State
 ├ CompositeRegion
 │ ├ State
 │ └ State
 └ OrthogonalRegion
   ╟ CompositeRegion
   ║ ├ State
   ║ └ State
   ╙ State
```

Library Interface for Hierarchical State Machines :: Transitions Rules

---

Whenever a state / region is activated:

1. All parents of the newly activated state also become active
2. For an active composite region - initial sub-state is activated
3. For an active orthogonal region - all sub-states get activated

---

Example:

```
[Root]                    // implicitly active
  ├ State
  ├ CompositeRegion
  │ ├ State
  │ └ State
  └ OrthogonalRegion
    ╟ CompositeRegion
    ║ ├ State
    ║ └ State
    ╙ State
```

Library Interface for Hierarchical State Machines :: Transitions Rules

---

Whenever a state / region is activated:

1. All parents of the newly activated state also become active
2. For an active composite region - initial sub-state is activated
3. For an active orthogonal region - all sub-states get activated

---

Example:

```
[Root]                      // implicitly active
  ├ State
  ├ CompositeRegion
  │   ├ State
  │   └ State
  └ [OrthogonalRegion]      // transition target
      ╟ CompositeRegion
      ║   ├ State
      ║   └ State
      ╙ State
```

Library Interface for Hierarchical State Machines :: Transitions Rules

---

Whenever a state / region is activated:

1. All parents of the newly activated state also become active
2. For an active composite region - initial sub-state is activated
3. For an active orthogonal region - all sub-states get activated

---

Example:

```
[Root]                      // implicitly active
  ├ State
  ├ CompositeRegion
  │ ├ State
  │ └ State
  └ [OrthogonalRegion]      // transition target
      ╟ [CompositeRegion]   // both sub-state 1
      ║ ├ State
      ║ └ State
      ╙ [State]             // and sub-state 2
```

Library Interface for Hierarchical State Machines :: Transitions Rules

---

Whenever a state / region is activated:

1. All parents of the newly activated state also become active
2. For an active composite region - initial sub-state is activated
3. For an active orthogonal region - all sub-states get activated

---

Example:

```
[Root]                      // implicitly active
  ├ State
  ├ CompositeRegion
  │   ├ State
  │   └ State
  └ [OrthogonalRegion]      // transition target
      ╟ [CompositeRegion]   // both sub-state 1
      ║   ├ State
      ║   └ [State]         // initial sub-state
      ╙ [State]             // and sub-state 2
```

Library Interface for Hierarchical State Machines :: Transitions Rules

---

Whenever a state / region is activated:

1. All parents of the newly activated state also become active
2. For an active composite region - initial sub-state is activated
3. For an active orthogonal region - all sub-states get activated

---

Example:

```
[Root]                    // implicitly active
  ├ State
  ├ CompositeRegion
  │  ├ State
  │  └ State
  └ [OrthogonalRegion]    // transition target
     ╟ [CompositeRegion]  // both sub-state 1
     ║  ├ State
     ║  └ [State]         // initial sub-state
     ╙ [State]            // and sub-state 2
```

```
// short form:
[Root]
  └ [OrthogonalRegion]
     ╟ [CompositeRegion]
     ║  └ [State]
     ╙ [State]
```

```
void SomeState::transition(Control& c, Context& _, const Time t) {






}
```

```
void SomeState::transition(Control& c, Context& _, const Time t) {
    // activate state
    control.changeTo<SomeOtherState>();







}
```

Library Interface for Hierarchical State Machines :: 3 Transitions

```
void SomeState::transition(Control& c, Context& _, const Time t) {
    // activate state
    control.changeTo<SomeOtherState>();

    // resume region’s state we left before (~UML's "history" pseudo-state)
    control.resume<ACompositeRegion>();



}
```

```
void SomeState::transition(Control& c, Context& _, const Time t) {
    // activate state
    control.changeTo<SomeOtherState>();

    // resume region’s state we left before (~UML's "history" pseudo-state)
    control.resume<ACompositeRegion>();

    // change the state to resume in the future
    control.schedule<SomeOtherState>();
}
```

Library Interface for Hierarchical State Machines :: Transitions within Hierarchy

```
[Root]
 ├ [Composite_1]
 │  ├ State_11
 │  └ [State_12]
 └ Orthogonal_2
    ╟ Composite_21
    ║  ├ State_211
    ║  └ State_212
    ╙ State_22
```

Library Interface for Hierarchical State Machines :: Transitions within Hierarchy

```
[Root]
 ├ [Composite_1]
 │  ├ State_11
 │  └ [State_12]      // changeTo() ─┐
 └ Orthogonal_2                      │
    ╟ Composite_21    // <───────────┘
    ║  ├ State_211
    ║  └ State_212
    ╙ State_22
```

Library Interface for Hierarchical State Machines :: Transitions within Hierarchy

```
[Root]
 ├ [Composite_1]
 │  ├ State_11
 │  └ [State_12]      // changeTo() ─┐
 └ Orthogonal_2                      │
    ╟ Composite_21    // <───────────┘
    ║  ├ State_211
    ║  └ State_212
    ╙ State_22
```

```
[Root]
 ├ Composite_1
 │  ├ State_11
 │  └ State_12
 └ [Orthogonal_2]      // parent activated
    ╟ [Composite_21]   // original target
    ║  ├ [State_211]   // initial sub-state activated
    ║  └ State_212
    ╙ [State_22]       // orthogonal state too
```

Library Interface for Hierarchical State Machines :: Transitions within Hierarchy

```
[Root]
 ├ [Composite_1]
 │  ├ State_11
 │  └ [State_12]      // changeTo() ─┐
 └ Orthogonal_2                     │
    ╟ Composite_21    // <──────────┘
    ║  ├ State_211
    ║  └ State_212
    ╙ State_22
```

```
[Root]
 ├ Composite_1
 │  ├ State_11
 │  └ State_12
 └ [Orthogonal_2]       // parent activated
    ╟ [Composite_21]    // original target
    ║  ├ [State_211]    // initial sub-state activated
    ║  └ State_212
    ╙ [State_22]        // orthogonal state too
```

CALL SEQUENCE:

Library Interface for Hierarchical State Machines :: Transitions within Hierarchy

```
[Root]
 ├ [Composite_1]
 │  ├ State_11
 │  └ [State_12]      // changeTo() ─┐
 └ Orthogonal_2                      │
    ╟ Composite_21    // <───────────┘
    ║  ├ State_211
    ║  └ State_212
    ╙ State_22
```

```
[Root]
 ├ Composite_1
 │  ├ State_11
 │  └ State_12
 └ [Orthogonal_2]        // parent activated
    ╟ [Composite_21]     // original target
    ║  ├ [State_211]     // initial sub-state activated
    ║  └ State_212
    ╙ [State_22]         // orthogonal state too
```

CALL SEQUENCE:

```
Root.update()
  Composite_1.update()
  Composite_1.transition()
    State_12.update()
  State_22.update()
```

Library Interface for Hierarchical State Machines :: Transitions within Hierarchy

```
[Root]                                           [Root]
 ├ [Composite_1]                                  ├ Composite_1
 │  ├ State_11                                    │  ├ State_11
 │  └ [State_12]     // changeTo() ─┐             │  └ State_12
 └ Orthogonal_2                     │             └ [Orthogonal_2]     // parent activated
    ╟ Composite_21   // <───────────┘                ╟ [Composite_21]  // original target
    ║  ├ State_211                                   ║  ├ [State_211]  // initial sub-state activated
    ║  └ State_212                                   ║  └ State_212
    ╙ State_22                                       ╙ [State_22]      // orthogonal state too
```

CALL SEQUENCE:

```
Root.update()
  Composite_1.update()
  Composite_1.transition()
    State_12.update()
  State_22.update()

  Orthogonal_2.substitute()
    Composite_21.substitute()
      State_211.substitute()
    State_22.substitute()
```

Library Interface for Hierarchical State Machines :: Transitions within Hierarchy

```
[Root]                                                    [Root]
 ├ [Composite_1]                                           ├ Composite_1
 │  ├ State_11                                             │  ├ State_11
 │  └ [State_12]      // changeTo() ─┐                     │  └ State_12
 └ Orthogonal_2                     │                     └ [Orthogonal_2]      // parent activated
    ╟ Composite_21    // <───────────┘                        ╟ [Composite_21]  // original target
    ║  ├ State_211                                            ║  ├ [State_211]  // initial sub-state activated
    ║  └ State_212                                            ║  └ State_212
    ╙ State_22                                                ╙ [State_22]      // orthogonal state too
```

CALL SEQUENCE:

```
Root.update()
  Composite_1.update()                       State_12.leave()
  Composite_1.transition()                 Composite_1.leave()
    State_12.update()
  State_22.update()

  Orthogonal_2.substitute()
    Composite_21.substitute()
      State_211.substitute()
    State_22.substitute()
```

Library Interface for Hierarchical State Machines :: Transitions within Hierarchy

```
[Root]
 ├ [Composite_1]
 │  ├ State_11
 │  └ [State_12]      // changeTo() ─┐
 └ Orthogonal_2                      │
    ╟ Composite_21    // <───────────┘
    ║  ├ State_211
    ║  └ State_212
    ╙ State_22
```

```
[Root]
 ├ Composite_1
 │  ├ State_11
 │  └ State_12
 └ [Orthogonal_2]       // parent activated
    ╟ [Composite_21]    // original target
    ║  ├ [State_211]    // initial sub-state activated
    ║  └ State_212
    ╙ [State_22]        // orthogonal state too
```

CALL SEQUENCE:

```
Root.update()
  Composite_1.update()
  Composite_1.transition()
    State_12.update()
//  State_12.transition()

  Orthogonal_2.substitute()
    Composite_21.substitute()
      State_211.substitute()
    State_22.substitute()
```

```
  State_12.leave()
Composite_1.leave()

Orthogonal_2.enter()
  Composite_21.enter()
    State_211.enter()
```

Library Interface for Hierarchical State Machines :: Transitions within Hierarchy

```
[Root]
 ├ [Composite_1]
 │  ├ State_11
 │  └ [State_12]      // changeTo() ─┐
 └ Orthogonal_2                      │
    ╟ Composite_21    // <───────────┘
    ║  ├ State_211
    ║  └ State_212
    ╙ State_22
```

```
[Root]
 ├ Composite_1
 │  ├ State_11
 │  └ State_12
 └ [Orthogonal_2]       // parent activated
    ╟ [Composite_21]    // original target
    ║  ├ [State_211]    // initial sub-state activated
    ║  └ State_212
    ╙ [State_22]        // orthogonal state too
```

CALL SEQUENCE:

```
Root.update()
  Composite_1.update()
  Composite_1.transition()
    State_12.update()
//  State_12.transition()

  Orthogonal_2.substitute()
    Composite_21.substitute()
      State_211.substitute()
    State_22.substitute()
```

```
  State_12.leave()
Composite_1.leave()

Orthogonal_2.enter()
  Composite_21.enter()
    State_211.enter()
  State_22.enter ()
```

```
Root.update()
  Orthogonal_2.update()
  Orthogonal_2.transition()
    Composite_21.update()
    Composite_21.transition()
      State_211.update()
      State_211.transition()
    State_22.update()
    State_22.transition()
```

Library Interface for Hierarchical State Machines :: Transitions within Hierarchy

```
[Root]
 ├ [Composite_1]
 │  ├ State_11
 │  └ [State_12]
 └ Orthogonal_2
    ╟ Composite_21
    ║  ├ State_211
    ║  └ State_212
    ╙ State_22
```

1. State_12.changeTo<Composite_21>()

```
[Root]
 ├ [Composite_1]
 │  ├ State_11
 │  └ [State_12]    // change to ─╮
 └ Orthogonal_2                   │
    ╟ Composite_21 // <───────────╯
    ║  ├ State_211
    ║  └ State_212
    ╙ State_22
```

1. State_12.changeTo<Composite_21>()

```
[Root]                                      [Root]
 ├ [Composite_1]                             ├ Composite_1
 │   ├ State_11                              │   ├ State_11
 │   └ [State_12]    // change to ┐          │   └ State_12
 └ Orthogonal_2                   │          └ [Orthogonal_2]
     ╟ Composite_21 // <──────────┘              ╟ [Composite_21]
     ║   ├ State_211                             ║   ├ [State_211]
     ║   └ State_212                             ║   └ State_212
     ╙ State_22                                  ╙ [State_22]
```

Library Interface for Hierarchical State Machines :: Transitions within Hierarchy

1. State_12.changeTo<Composite_21>()
2. **Orthogonal_2.resume<Composite_1>()**

```
[Root]                                          [Root]
  ├ [Composite_1]                                  ├ Composite_1    // <─────────┐
  │   ├ State_11                                   │   ├ State_11                │
  │   └ [State_12]    // change to ┐               │   └ State_12                │
  └ Orthogonal_2                   │               └ [Orthogonal_2] // resume ───┘
      ╟ Composite_21 // <──────────┘                   ╟ [Composite_21]
      ║   ├ State_211                                  ║   ├ [State_211]
      ║   └ State_212                                  ║   └ State_212
      ╙ State_22                                       ╙ [State_22]
```

Library Interface for Hierarchical State Machines :: Transitions within Hierarchy

1. State_12.changeTo<Composite_21>()
2. Orthogonal_2.resume<Composite_1>()

```
[Root]                                      [Root]                                     [Root]
 ├ [Composite_1]                             ├ Composite_1   // <─────────┐             ├ [Composite_1]
 │  ├ State_11                               │  ├ State_11                │             │  ├ State_11
 │  └ [State_12]     // change to ┐          │  └ State_12                │             │  └ [State_12]
 └ Orthogonal_2                   │          └ [Orthogonal_2] // resume ──┘             └ Orthogonal_2
    ╟ Composite_21 // <───────────┘             ╟ [Composite_21]                           ╟ Composite_21
    ║  ├ State_211                              ║  ├ [State_211]                           ║  ├ State_211
    ║  └ State_212                              ║  └ State_212                             ║  └ State_212
    ╙ State_22                                  ╙ [State_22]                               ╙ State_22
```

Library Interface for Hierarchical State Machines :: Transitions within Hierarchy

1. State_12.changeTo<Composite_21>()
2. Orthogonal_2.resume<Composite_1>()

```
[Root]
 ├ [Composite_1]
 │  ├ State_11
 │  └ [State_12]     // change to ╮
 └ Orthogonal_2                   │
    ╟ Composite_21 // <───────────╯
    ║  ├ State_211
    ║  └ State_212
    ╙ State_22
```

```
[Root]
 ├ Composite_1    // <──────╮
 │  ├ State_11              │
 │  └ State_12              │
 └ [Orthogonal_2] // resume ╯
    ╟ [Composite_21]
    ║  ├ [State_211]
    ║  └ State_212
    ╙ [State_22]
```

```
[Root]
 ├ [Composite_1] // original target
 │  ├ State_11
 │  └ [State_12] // resumed sub-s.
 └ Orthogonal_2
    ╟ Composite_21
    ║  ├ State_211
    ║  └ State_212
    ╙ State_22
```

Library Interface for Hierarchical State Machines :: Transitions within Hierarchy

```
[Root]
  ├ [Composite_1]
  │   ├ State_11
  │   └ [State_12]
  └ Orthogonal_2
      ╟ Composite_21
      ║   ├ State_211
      ║   └ State_212
      ╙ State_22
```

1. State_12.changeTo<Composite_21>()

```
[Root]
 ├ [Composite_1]
 │   ├ State_11
 │   └ [State_12]    // change to ─╮
 └ Orthogonal_2                    │
     ╟ Composite_21 // <───────────╯
     ║   ├ State_211
     ║   └ State_212
     ╙ State_22
```

Library Interface for Hierarchical State Machines :: Transitions within Hierarchy

1. State_12.changeTo<Composite_21>()

```
[Root]
 ├ [Composite_1]
 │   ├ State_11
 │   └ [State_12]     // change to ─╮
 └ Orthogonal_2                     │
     ╟ Composite_21 // <────────────╯
     ║   ├ State_211
     ║   └ State_212
     ╙ State_22
```

```
[Root]
 ├ Composite_1
 │   ├ State_11
 │   └ State_12
 └ [Orthogonal_2]
     ╟ [Composite_21]
     ║   ├ [State_211]
     ║   └ State_212
     ╙ [State_22]
```

Library Interface for Hierarchical State Machines :: Transitions within Hierarchy

1. State_12.changeTo<Composite_21>()
2. **Orthogonal_2.schedule<State_11>()**

```
[Root]
  ├ [Composite_1]
  │   ├ State_11
  │   └ [State_12]     // change to ─┐
  └ Orthogonal_2                     │
      ╟ Composite_21 // <────────────┘
      ║   ├ State_211
      ║   └ State_212
      ╙ State_22
```

```
[Root]
  ├ Composite_1
  │   ├ State_11  // <───────┐
  │   └ State_12             │
  └ [Orthogonal_2] // schedule ┘
      ╟ [Composite_21]
      ║   ├ [State_211]
      ║   └ State_212
      ╙ [State_22]
```

Library Interface for Hierarchical State Machines :: Transitions within Hierarchy

1. `State_12.changeTo<Composite_21>()`
2. `Orthogonal_2.schedule<State_11>()`

```
[Root]
  ├ [Composite_1]
  │   ├ State_11
  │   └ [State_12]     // change to ─┐
  └ Orthogonal_2                     │
      ╟ Composite_21 // <────────────┘
      ║   ├ State_211
      ║   └ State_212
      ╙ State_22
```

```
[Root]
  ├ Composite_1
  │   ├ State_11  // <───────────┐
  │   └ State_12                 │
  └ [Orthogonal_2] // schedule ──┘
      ╟ [Composite_21]
      ║   ├ [State_211]
      ║   └ State_212
      ╙ [State_22]
```

```
[Root]
  ├ Composite_1
  │   ├ State_11
  │   └ State_12
  └ [Orthogonal_2]
      ╟ [Composite_21]
      ║   ├ [State_211]
      ║   └ State_212
      ╙ [State_22]
```

Library Interface for Hierarchical State Machines :: Transitions within Hierarchy

1. State_12.changeTo<Composite_21>()
2. Orthogonal_2.schedule<State_11>()
3. **Orthogonal_2.resume<Composite_1>()**

```
[Root]
 ├ [Composite_1]
 │  ├ State_11
 │  └ [State_12]     // change to ─┐
 └ Orthogonal_2                    │
    ╟ Composite_21 // <────────────┘
    ║  ├ State_211
    ║  └ State_212
    ╙ State_22
```

```
[Root]
 ├ Composite_1
 │  ├ State_11  // <──────────┐
 │  └ State_12                │
 └ [Orthogonal_2] // schedule ┘
    ╟ [Composite_21]
    ║  ├ [State_211]
    ║  └ State_212
    ╙ [State_22]
```

```
[Root]
 ├ Composite_1      // <───────┐
 │  ├ State_11                 │
 │  └ State_12                 │
 └ [Orthogonal_2]   // resume ─┘
    ╟ [Composite_21]
    ║  ├ [State_211]
    ║  └ State_212
    ╙ [State_22]
```

Library Interface for Hierarchical State Machines :: Transitions within Hierarchy

1. State_12.changeTo<Composite_21>()
2. Orthogonal_2.schedule<State_11>()
3. Orthogonal_2.resume<Composite_1>()

```
[Root]
 ├ [Composite_1]
 │  ├ State_11
 │  └ [State_12]     // change to ─┐
 └ Orthogonal_2                    │
    ╟ Composite_21 // <────────────┘
    ║  ├ State_211
    ║  └ State_212
    ╙ State_22
```

```
[Root]
 ├ Composite_1
 │  ├ State_11  // <──────────┐
 │  └ State_12                │
 └ [Orthogonal_2] // schedule ┘
    ╟ [Composite_21]
    ║  ├ [State_211]
    ║  └ State_212
    ╙ [State_22]
```

```
[Root]
 ├ Composite_1      // <──────────┐
 │  ├ State_11                    │
 │  └ State_12                    │
 └ [Orthogonal_2]   // resume ────┘
    ╟ [Composite_21]
    ║  ├ [State_211]
    ║  └ State_212
    ╙ [State_22]
```

```
[Root]
 ├ [Composite_1]
 │  ├ [State_11]
 │  └ State_12
 └ Orthogonal_2
    ╟ Composite_21
    ║  ├ State_211
    ║  └ State_212
    ╙ State_22
```

Library Interface for Hierarchical State Machines :: Transitions within Hierarchy

1. State_12.changeTo<Composite_21>()
2. Orthogonal_2.schedule<State_11>()
3. Orthogonal_2.resume<Composite_1>()

```
[Root]
 ├ [Composite_1]
 │  ├ State_11
 │  └ [State_12]     // change to ┐
 └ Orthogonal_2                   │
    ╟ Composite_21 // <───────────┘
    ║  ├ State_211
    ║  └ State_212
    ╙ State_22
```

```
[Root]
 ├ Composite_1
 │  ├ State_11  // <──────┐
 │  └ State_12            │
 └ [Orthogonal_2] // schedule
    ╟ [Composite_21]
    ║  ├ [State_211]
    ║  └ State_212
    ╙ [State_22]
```

```
[Root]
 ├ Composite_1       // <───────┐
 │  ├ State_11                  │
 │  └ State_12                  │
 └ [Orthogonal_2]    // resume ─┘
    ╟ [Composite_21]
    ║  ├ [State_211]
    ║  └ State_212
    ╙ [State_22]
```

```
[Root]
 ├ [Composite_1] // original target
 │  ├ [State_11] // scheduled sub-state
 │  └ State_12
 └ Orthogonal_2
    ╟ Composite_21
    ║  ├ State_211
    ║  └ State_212
    ╙ State_22
```

AGENDA

* INTRO
  * A Library for Video Games
  * High Level Design Overview

* SIMPLE STATE MACHINES
  * Example - Player Character
  * Library Interface for Simple State Machines
  * Example - Player Character, FSM

* HIERARCHICAL STATE MACHINES
  * Example - Complex Player Character
  * Library Interface for Hierarchical State Machines
  * **Example - Complex Player Character, FSM**
  * State Machine Complexity Intuition

* OUTRO
  * Advanced Library Interface
  * Post-Mortem
  * Future Work

* SUMMARY

Example - Complex Player Character, FSM :: State Diagram

```
Root
 ├ Walking
 │  ├ Idle
 │  ├ Firing
 │  └ Reloading
 └ Sprinting
```

Example - Complex Player Character, FSM :: State Diagram

```
Root
 ├ Walking
 │  ├ Idle
 │  ├ Firing
 │  └ Reloading
 └ Sprinting
    ├ Idle
    └ Reloading
```

Notice how invalid state (Sprinting-Firing) is now impossible

---

With plain state variables Sprinting-Firing was a possible but invalid combination:

Example - Complex Player Character, FSM :: State Diagram

```
Root
 ├ Walking
 │  ├ Idle
 │  ├ Firing
 │  └ Reloading
 └ Sprinting
    ├ Idle
    └ Reloading
```

Notice how invalid state (Sprinting-Firing) is now impossible

---

With plain state variables Sprinting-Firing was a possible but invalid combination:

	Idle	**Firing**	**Reloading**
Walking	√	√	√
Sprinting	√	×	√

Example - Complex Player Character, FSM :: C++ Pseudo-Code

```
#include <hfsm.h>

struct Context { /* ... */ };

using M = hfsm::Machine<Context>;      // sugar
```

Example - Complex Player Character, FSM :: C++ Pseudo-Code

```
#include <hfsm.h>

struct Context { /* ... */ };

using M = hfsm::Machine<Context>;      // sugar

struct Walking : M::Base {
    struct Idle : M::Base { /* ... */ };
    struct Firing : M::Timed { /* ... */ };
    struct Reloading : M::Timed { /* ... */ };
}
```

Example - Complex Player Character, FSM :: C++ Pseudo-Code

```cpp
#include <hfsm.h>

struct Context { /* ... */ };

using M = hfsm::Machine<Context>;      // sugar

struct Walking : M::Base {
    struct Idle : M::Base { /* ... */ };
    struct Firing : M::Timed { /* ... */ };
    struct Reloading : M::Timed { /* ... */ };
}

struct Sprinting : M::Composite {
    struct Idle : M::Base { /* ... */ };
    struct Reloading : M::Timed { /* ... */ };
}
```

Example - Complex Player Character, FSM :: C++ Pseudo-Code

```cpp
#include <hfsm.h>

struct Context { /* ... */ };

using M = hfsm::Machine<Context>;     // sugar

struct Walking : M::Base {
    struct Idle : M::Base { /* ... */ };
    struct Firing : M::Timed { /* ... */ };
    struct Reloading : M::Timed { /* ... */ };
}

struct Sprinting : M::Composite {
    struct Idle : M::Base { /* ... */ };
    struct Reloading : M::Timed { /* ... */ };
}
```

```cpp
using PlayerFSM = M::CompositeRoot<
    M::Composite<Walking,
        M::State<Idle>,
        M::State<Firing>,
        M::State<Reloading>
    >,
    M::Composite<Sprinting,
        M::State<Idle>,
        M::State<Reloading>
    >
>;
```

Example - Complex Player Character, FSM :: C++ Pseudo-Code

```cpp
#include <hfsm.h>

struct Context { /* ... */ };

using M = hfsm::Machine<Context>;      // sugar

struct Walking : M::Base {
    struct Idle : M::Base { /* ... */ };
    struct Firing : M::Timed { /* ... */ };
    struct Reloading : M::Timed { /* ... */ };
}

struct Sprinting : M::Composite {
    struct Idle : M::Base { /* ... */ };
    struct Reloading : M::Timed { /* ... */ };
}
```

```cpp
using PlayerFSM = M::CompositeRoot<
    M::Composite<Walking,
        M::State<Idle>,
        M::State<Firing>,
        M::State<Reloading>
    >,
    M::Composite<Sprinting,
        M::State<Idle>,
        M::State<Reloading>
    >
>;

void start() {
    Context c;
    PlayerFSM fsm(c);

    fsm.enter();
}
```

Example - Complex Player Character, FSM :: C++ Pseudo-Code

```cpp
#include <hfsm.h>

struct Context { /* ... */ };

using M = hfsm::Machine<Context>;      // sugar

struct Walking : M::Base {
    struct Idle : M::Base { /* ... */ };
    struct Firing : M::Timed { /* ... */ };
    struct Reloading : M::Timed { /* ... */ };
}

struct Sprinting : M::Composite {
    struct Idle : M::Base { /* ... */ };
    struct Reloading : M::Timed { /* ... */ };
}
```

```cpp
using PlayerFSM = M::CompositeRoot<
    M::Composite<Walking,
        M::State<Idle>,
        M::State<Firing>,
        M::State<Reloading>
    >,
    M::Composite<Sprinting,
        M::State<Idle>,
        M::State<Reloading>
    >
>;

void start() {
    Context c;
    PlayerFSM fsm(c);

    fsm.enter();
}

void update(const float detlaTime) {
    fsm.update(detlaTime);
}
```

Example - Complex Player Character, FSM :: C++ Pseudo-Code

```
struct Walking : M::Base {

};
```

```
struct Sprinting : M::Base {

};
```

Example - Complex Player Character, FSM :: C++ Pseudo-Code

```
struct Walking : M::Base {
    void enter(Context& _, const Time t) {
        playWalkAnimation();
    }
};
```

```
struct Sprinting : M::Base {
};
```

Example - Complex Player Character, FSM :: C++ Pseudo-Code

```cpp
struct Walking : M::Base {
    void enter(Context& _, const Time t) {
        playWalkAnimation();
    }

    void transition(Control& c, Context& _,
                    const Time t)
    {
        if (keyPressed(KeySprint))
            c.changeTo<Sprinting>();
    }
};
```

```cpp
struct Sprinting : M::Base {
};
```

Example - Complex Player Character, FSM :: C++ Pseudo-Code

```
struct Walking : M::Base {
    void enter(Context& _, const Time t) {
        playWalkAnimation();
    }

    void transition(Control& c, Context& _,
                    const Time t)
    {
        if (keyPressed(KeySprint))
            c.changeTo<Sprinting>();
    }

    void update(Context& _, const Time t) {
        processWalkMovement(t);
    }


};
```

```
struct Sprinting : M::Base {


















};
```

Example - Complex Player Character, FSM :: C++ Pseudo-Code

```
struct Walking : M::Base {
    void enter(Context& _, const Time t) {
        playWalkAnimation();
    }

    void transition(Control& c, Context& _,
                    const Time t)
    {
        if (keyPressed(KeySprint))
            c.changeTo<Sprinting>();
    }

    void update(Context& _, const Time t) {
        processWalkMovement(t);
    }

    struct Idle : M::Base { /* ... */ };
};
```

```
struct Sprinting : M::Base {









};
```

Example - Complex Player Character, FSM :: C++ Pseudo-Code

```cpp
struct Walking : M::Base {
    void enter(Context& _, const Time t) {
        playWalkAnimation();
    }

    void transition(Control& c, Context& _,
                    const Time t)
    {
        if (keyPressed(KeySprint))
            c.changeTo<Sprinting>();
    }

    void update(Context& _, const Time t) {
        processWalkMovement(t);
    }

    struct Idle : M::Base { /* ... */ };
};
```

```cpp
struct Sprinting : M::Base {
    void enter(Context& _, const Time t) {
        playSprintAnimation();
    }

};
```

Example - Complex Player Character, FSM :: C++ Pseudo-Code

```
struct Walking : M::Base {
    void enter(Context& _, const Time t) {
        playWalkAnimation();
    }

    void transition(Control& c, Context& _,
                    const Time t)
    {
        if (keyPressed(KeySprint))
            c.changeTo<Sprinting>();
    }

    void update(Context& _, const Time t) {
        processWalkMovement(t);
    }

    struct Idle : M::Base { /* ... */ };
};
```

```
struct Sprinting : M::Base {
    void enter(Context& _, const Time t) {
        playSprintAnimation();
    }

    void transition(Control& c, Context& _,
                    const Time t)
    {
        if (!keyPressed(KeySprint))
            c.changeTo<Walking>();
    }
};
```

Example - Complex Player Character, FSM :: C++ Pseudo-Code

```cpp
struct Walking : M::Base {
    void enter(Context& _, const Time t) {
        playWalkAnimation();
    }

    void transition(Control& c, Context& _,
                    const Time t)
    {
        if (keyPressed(KeySprint))
            c.changeTo<Sprinting>();
    }

    void update(Context& _, const Time t) {
        processWalkMovement(t);
    }

    struct Idle : M::Base { /* ... */ };
};
```

```cpp
struct Sprinting : M::Base {
    void enter(Context& _, const Time t) {
        playSprintAnimation();
    }

    void transition(Control& c, Context& _,
                    const Time t)
    {
        if (!keyPressed(KeySprint))
            c.changeTo<Walking>();
    }

    void update(Context& _, const Time t) {
        processSprintMovement(t);
    }


};
```

Example - Complex Player Character, FSM :: C++ Pseudo-Code

```
struct Walking : M::Base {
    void enter(Context& _, const Time t) {
        playWalkAnimation();
    }

    void transition(Control& c, Context& _,
                    const Time t)
    {
        if (keyPressed(KeySprint))
            c.changeTo<Sprinting>();
    }

    void update(Context& _, const Time t) {
        processWalkMovement(t);
    }

    struct Idle : M::Base { /* ... */ };
};
```

```
struct Sprinting : M::Base {
    void enter(Context& _, const Time t) {
        playSprintAnimation();
    }

    void transition(Control& c, Context& _,
                    const Time t)
    {
        if (!keyPressed(KeySprint))
            c.changeTo<Walking>();
    }

    void update(Context& _, const Time t) {
        processSprintMovement(t);
    }

    struct Idle : M::Base { /* ... */ };
};
```

Example - Complex Player Character, FSM :: C++ Pseudo-Code

```cpp
struct Walking : M::Base {
    void enter(Context& _, const Time t) {
        playWalkAnimation();
    }

    void transition(Control& c, Context& _,
                    const Time t)
    {
        if (keyPressed(KeySprint))
            c.changeTo<Sprinting>();
    }

    void update(Context& _, const Time t) {
        processWalkMovement(t);
    }

    struct Idle : M::Base { /* ... */ };
};
```

```cpp
struct Sprinting : M::Base {
    void enter(Context& _, const Time t) {
        playSprintAnimation();
    }

    void transition(Control& c, Context& _,
                    const Time t)
    {
        if (!keyPressed(KeySprint))
            c.changeTo<Walking>();
    }

    void update(Context& _, const Time t) {
        processSprintMovement(t);
    }

    struct Idle : M::Base { /* ... */ };
};
```

Notice, how Walking and Sprinting only deal with movement, and never touch weapon operation

Example - Complex Player Character, FSM :: C++ Pseudo-Code

```
struct Walking : M::Base {
    void enter(Context& _, const Time t) {
        playWalkAnimation();
    }

    void transition(Control& c, Context& _,
                    const Time t)
    {
        if (keyPressed(KeySprint))
            c.changeTo<Sprinting>();
    }

    void update(Context& _, const Time t) {
        processWalkMovement(t);
    }

    struct Idle : M::Base { /* ... */ };
};
```

```
struct Sprinting : M::Base {
    void enter(Context& _, const Time t) {
        playSprintAnimation();
    }

    void transition(Control& c, Context& _,
                    const Time t)
    {
        if (!keyPressed(KeySprint))
            c.changeTo<Walking>();
    }

    void update(Context& _, const Time t) {
        processSprintMovement(t);
    }

    struct Idle : M::Base { /* ... */ };
};
```

Notice, how Walking and Sprinting only deal with movement, and never touch weapon operation

There's zero dependency between movement parent regions and weapon operation sub-regions

Example - Complex Player Character, FSM :: C++ Pseudo-Code

```
struct Walking : M::Base {
    void enter(Context& _, const Time t) {
        playWalkAnimation();
    }

    void transition(Control& c, Context& _,
                    const Time t)
    {
        if (keyPressed(KeySprint))
            c.changeTo<Sprinting>();
    }

    void update(Context& _, const Time t) {
        processWalkMovement(t);
    }

    struct Idle : M::Base { /* ... */ };
};
```

```
struct Sprinting : M::Base {
    void enter(Context& _, const Time t) {
        playSprintAnimation();
    }

    void transition(Control& c, Context& _,
                    const Time t)
    {
        if (!keyPressed(KeySprint))
            c.changeTo<Walking>();
    }

    void update(Context& _, const Time t) {
        processSprintMovement(t);
    }

    struct Idle : M::Base { /* ... */ };
};
```

Notice, how Walking and Sprinting only deal with movement, and never touch weapon operation
There's zero dependency between movement parent regions and weapon operation sub-regions
Perfect loose coupling!

Example - Complex Player Character, FSM :: C++ Pseudo-Code

```
struct Context {
    unsigned weaponAmmoCount;
    unsigned weaponAmmoCapacity;
    unsigned spareAmmoCount;
}
```

```
struct Sprinting : M::Base {
    struct Idle : M::Base {







    }
};
```

Example - Complex Player Character, FSM :: C++ Pseudo-Code

```
struct Context {
    unsigned weaponAmmoCount;
    unsigned weaponAmmoCapacity;
    unsigned spareAmmoCount;
}
```

```
struct Sprinting : M::Base {
    struct Idle : M::Base {
        void transition(Control& c, Context& _, const Time t) {









        }
    }
};
```

Example - Complex Player Character, FSM :: C++ Pseudo-Code

```
struct Context {
    unsigned weaponAmmoCount;
    unsigned weaponAmmoCapacity;
    unsigned spareAmmoCount;
}
```

```
struct Sprinting : M::Base {
    struct Idle : M::Base {
        void transition(Control& c, Context& _, const Time t) {
            if (_.weaponAmmoCount > 0 && keyPressed(KeyFire))
                c.changeTo<Firing>();









        }
    }
};
```

Example - Complex Player Character, FSM :: C++ Pseudo-Code

```cpp
struct Context {
    unsigned weaponAmmoCount;
    unsigned weaponAmmoCapacity;
    unsigned spareAmmoCount;
}
```

```cpp
struct Sprinting : M::Base {
    struct Idle : M::Base {
        void transition(Control& c, Context& _, const Time t) {
            if (_.weaponAmmoCount > 0 && keyPressed(KeyFire))
                c.changeTo<Firing>();
            else if (
                    _.weaponAmmoCount < _.weaponAmmoCapacity &&
                    _.spareAmmoCount > 0 &&
                    (keyPressed(KeyReload) || _.weaponAmmoCount == 0))
            {
                c.changeTo<Reloading>();
            }



        }
    }
};
```

Example - Complex Player Character, FSM :: C++ Pseudo-Code

```
struct Context {
    unsigned weaponAmmoCount;
    unsigned weaponAmmoCapacity;
    unsigned spareAmmoCount;
}
```

```
struct Sprinting : M::Base {
    struct Idle : M::Base {
        void transition(Control& c, Context& _, const Time t) {
            if (_.weaponAmmoCount > 0 && keyPressed(KeyFire))
                c.changeTo<Firing>();
            else if (havePerk(PerkSprintReload) &&
                     _.weaponAmmoCount < _.weaponAmmoCapacity &&
                     _.spareAmmoCount > 0 &&
                     (keyPressed(KeyReload) || _.weaponAmmoCount == 0))
            {
                c.changeTo<Reloading>();
            }



        }
    }
};
```

Example - Complex Player Character, FSM :: C++ Pseudo-Code

```
struct Context {
    unsigned weaponAmmoCount;
    unsigned weaponAmmoCapacity;
    unsigned spareAmmoCount;
}
```

```
struct Sprinting : M::Base {
    struct Idle : M::Base {
        void transition(Control& c, Context& _, const Time t) {
            if (_.weaponAmmoCount > 0 && keyPressed(KeyFire))
                c.changeTo<Firing>();
            else if (havePerk(PerkSprintReload) &&
                     _.weaponAmmoCount < _.weaponAmmoCapacity &&
                     _.spareAmmoCount > 0 &&
                     (keyPressed(KeyReload) || _.weaponAmmoCount == 0))
            {
                c.changeTo<Reloading>();
            }

//          void update(Context& c, const Time t) {
//              processMovement(t);
//          }
        }
    }
};
```

Spinting::Idle no longer has any movement-related logic, which moved to its parent region

Case Study: Endless Runner Player Character

```
Root
 ├ Alive                    (infinite forward run)
```

```
Root
  ├ Hurt        (hurt animation, lives reduction)
  └ Dead         (game over, fall to the ground)
```

Case Study: Endless Runner Player Character

```
Root
 ├ Alive                 (infinite forward run)
 │  ├ Detached           (player-controlled motion)
 │  └ Attached           (follow spline path)
```

```
Root
 ├ Hurt        (hurt animation, lives reduction)
 └ Dead        (game over, fall to the ground)
```

Case Study: Endless Runner Player Character

```
Root
 ├ Alive                            (infinite forward run)
 │  ├ Detached                  (player-controlled motion)
 │  │  ├ Gravitational          (XYZ components change)
 │  │  └ Dash                    (X-exclusive movement)
 │  └ Attached                       (follow spline path)
::::::::::::::::::::::::::::::::::::

Root
::::::::::::::::::::::::::::::::::::
  ├ Hurt          (hurt animation, lives reduction)
  └ Dead            (game over, fall to the ground)
```

Case Study: Endless Runner Player Character

```
Root
 ├ Alive                          (infinite forward run)
 │  ├ Detached              (player-controlled motion)
 │  │  ├ Gravitational    (XYZ components change)
 │  │  │  ╠ Vertical          (Y-component change)
 │  │  │  ╠ Horizontal            (X-speed change)
 │  │  │  ╚ Strafe            (Z-component change)
 │  │  └ Dash                (X-exclusive movement)
 │  └ Attached                 (follow spline path)

Root
 ├ Hurt         (hurt animation, lives reduction)
 └ Dead          (game over, fall to the ground)
```

Case Study: Endless Runner Player Character

```
Root
 ├ Alive                          (infinite forward run)
 │   ├ Detached               (player-controlled motion)
 │   │   ├ Gravitational    (XYZ components change)
 │   │   │   ╟ Vertical          (Y-component change)
 │   │   │   ╟ Horizontal            (X-speed change)
 │   │   │   ╙ Strafe            (Z-component change)
 │   │   └ Dash               (X-exclusive movement)
 │   │       ├ On
 │   │       └ Depleted
 │   └ Attached                  (follow spline path)

Root
 ├ Hurt         (hurt animation, lives reduction)
 └ Dead          (game over, fall to the ground)
```

Case Study: Endless Runner Player Character

```
Root
 ├ Alive                     (infinite forward run)
 │  ├ Detached            (player-controlled motion)
 │  │  ├ Gravitational   (XYZ components change)
 │  │  │  ╟ Vertical        (Y-component change)
 │  │  │  ║  ├ Fall
 │  │  │  ║  ├ Land
 │  │  │  ║  ├ Run
 │  │  │  ║  ├ JumpPad
 │  │  │  ║  ├ Jump
 │  │  │  ║  └ DoubleJump
 │  │  │  ╟ Horizontal         (X-speed change)
 │  │  │  ╙ Strafe          (Z-component change)
 │  │  └ Dash            (X-exclusive movement)
 │  │     ├ On
 │  │     └ Depleted
 │  └ Attached              (follow spline path)
```

```
Root
 ├ Hurt        (hurt animation, lives reduction)
 └ Dead         (game over, fall to the ground)
    ├ Ground
    ├ Fall
    └ Land
```

Case Study: Endless Runner Player Character

```
Root
 ├ Alive                          (infinite forward run)
 │  ├ Detached                    (player-controlled motion)
 │  │  ├ Gravitational            (XYZ components change)
 │  │  │  ├ Vertical              (Y-component change)
 │  │  │  │  ├ Fall
 │  │  │  │  ├ Land
 │  │  │  │  ├ Run
 │  │  │  │  ├ JumpPad
 │  │  │  │  ├ Jump
 │  │  │  │  └ DoubleJump
 │  │  │  ├ Horizontal            (X-speed change)
 │  │  │  │  ├ Normal
 │  │  │  │  ├ Boosted
 │  │  │  │  └ Inertia
 │  │  │  └ Strafe                (Z-component change)
 │  │  └ Dash                     (X-exclusive movement)
 │  │     ├ On
 │  │     └ Depleted
 │  └ Attached                    (follow spline path)
```

```
Root
 ├ Hurt                 (hurt animation, lives reduction)
 └ Dead                 (game over, fall to the ground)
    ├ Ground
    ├ Fall
    └ Land
```

Case Study: Endless Runner Player Character

```
Root
 ├ Alive                        (infinite forward run)
 │  ├ Detached             (player-controlled motion)
 │  │  ├ Gravitational     (XYZ components change)
 │  │  │  ╟ Vertical             (Y-component change)
 │  │  │  ║  ├ Fall
 │  │  │  ║  ├ Land
 │  │  │  ║  ├ Run
 │  │  │  ║  ├ JumpPad
 │  │  │  ║  ├ Jump
 │  │  │  ║  └ DoubleJump
 │  │  │  ╟ Horizontal                (X-speed change)
 │  │  │  ║  ├ Normal
 │  │  │  ║  ├ Boosted
 │  │  │  ║  └ Inertia
 │  │  │  ╙ Strafe               (Z-component change)
 │  │  │     ├ None
 │  │  │     ├ Left
 │  │  │     └ Right
 │  │  └ Dash                 (X-exclusive movement)
 │  │     ├ On
 │  │     └ Depleted
 │  └ Attached                   (follow spline path)
┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄
```

```
Root
┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄
  ├ Hurt         (hurt animation, lives reduction)
  └ Dead           (game over, fall to the ground)
     ├ Ground
     ├ Fall
     └ Land
```

AGENDA

* INTRO
  * A Library for Video Games
  * High Level Design Overview

* SIMPLE STATE MACHINES
  * Example - Player Character
  * Library Interface for Simple State Machines
  * Example - Player Character, FSM

* HIERARCHICAL STATE MACHINES
  * Example - Complex Player Character
  * Library Interface for Hierarchical State Machines
  * Example - Complex Player Character, FSM
  * **State Machine Complexity Intuition**

* OUTRO
  * Advanced Library Interface
  * Post-Mortem
  * Future Work

* SUMMARY

State Machine Complexity Intuition

There’s a math concept that resembles a feature in terms of complexity and interaction:

State Machine Complexity Intuition

There's a math concept that resembles a feature in terms of complexity and interaction:

* **State Variable** ~ **Number**

State Machine Complexity Intuition

There's a math concept that resembles a feature in terms of complexity and interaction:

* State Variable	~ Number
* **Feature**	~ **Matrix**

State Machine Complexity Intuition

There's a math concept that resembles a feature in terms of complexity and interaction:

* State Variable ~ Number
* Feature ~ Matrix
* **Conditional Expression on State Variables ~ Numeric Product**

State Machine Complexity Intuition

There's a math concept that resembles a feature in terms of complexity and interaction:

* State Variable ~ Number
* Feature ~ Matrix
* Conditional Expression on State Variables ~ Numeric Product
* **Feature Composition / Interaction ~ Matrix Multiplication**

State Machine Complexity Intuition

There's a math concept that resembles a feature in terms of complexity and interaction:

* State Variable ~ Number
* Feature ~ Matrix
* Conditional Expression on State Variables ~ Numeric Product
* Feature Composition / Interaction ~ Matrix Multiplication

Feature composition using plain state variables (feels like ☺):

State Machine Complexity Intuition

There's a math concept that resembles a feature in terms of complexity and interaction:

* State Variable ~ Number
* Feature ~ Matrix
* Conditional Expression on State Variables ~ Numeric Product
* Feature Composition / Interaction ~ Matrix Multiplication

Feature composition using plain state variables (feels like ☺):

$$\begin{bmatrix} a11 & a12 & a13 \\ a21 & a22 & a23 \end{bmatrix} \times \begin{bmatrix} b11 \\ b21 \\ b31 \end{bmatrix} = \begin{bmatrix} a11 \times b11 + a12 \times b21 + a13 \times b31 \\ a21 \times b11 + a22 \times b21 + a23 \times b31 \end{bmatrix}$$

// a11, b11 ~ State Variables

// a11 × b11 ~ Conditional Expression

State Machine Complexity Intuition

There's a math concept that resembles a feature in terms of complexity and interaction:

* State Variable ~ Number
* Feature ~ Matrix
* Conditional Expression on State Variables ~ Numeric Product
* Feature Composition / Interaction ~ Matrix Multiplication

Feature composition using plain state variables (feels like ☺):

$$\begin{bmatrix} a11 & a12 & a13 \\ a21 & a22 & a23 \end{bmatrix} \times \begin{bmatrix} b11 \\ b21 \\ b31 \end{bmatrix} = \begin{bmatrix} a11 \times b11 + a12 \times b21 + a13 \times b31 \\ a21 \times b11 + a22 \times b21 + a23 \times b31 \end{bmatrix}$$

// a11, b11 ~ State Variables

// a11 × b11 ~ Conditional Expression

Feature composition using FSM framework (feels like ☺):

State Machine Complexity Intuition

There's a math concept that resembles a feature in terms of complexity and interaction:

* Feature ~ Matrix
* State Variable ~ Matrix Component
* Conditional Expression on State Variables ~ Matrix Component Product
* Feature Composition / Interaction ~ Matrix Multiplication

Feature composition using plain state variables (feels like ☺):

$$\begin{bmatrix} a11 & a12 & a13 \\ a21 & a22 & a23 \end{bmatrix} \times \begin{bmatrix} b11 \\ b21 \\ b31 \end{bmatrix} = \begin{bmatrix} a11 \times b11 + a12 \times b21 + a13 \times b31 \\ a21 \times b11 + a22 \times b21 + a23 \times b31 \end{bmatrix}$$

// a11, b11 ~ State Variables

// a11 × b11 ~ Conditional Expression

Feature composition using FSM framework (feels like ☺):

$$A \times B = C$$

AGENDA

* INTRO
  * A Library for Video Games
  * High Level Design Overview

* SIMPLE STATE MACHINES
  * Example - Player Character
  * Library Interface for Simple State Machines
  * Example - Player Character, FSM

* HIERARCHICAL STATE MACHINES
  * Example - Complex Player Character
  * Library Interface for Hierarchical State Machines
  * Example - Complex Player Character, FSM
  * State Machine Complexity Intuition

* OUTRO
  * **Advanced Library Interface**
  * Post-Mortem
  * Future Work

* SUMMARY

Advanced Library Interface :: State Injections

```
// from HFSM test.cpp
struct B_2_2
    : M::TrackedBase
{
    void substitute(Control& control,
                    Context&, const Time) const
    {
        if (entryCount() == 2) // ?
            control.resume<A>();
    }
};
```

Advanced Library Interface :: State Injections

```cpp
// from HFSM test.cpp
struct B_2_2
    : M::TrackedBase // injection!
{
    void substitute(Control& control,
                    Context&, const Time) const
    {
        if (entryCount() == 2) // from M::Tracked
            control.resume<A>();
    }
};
```

```cpp
// from HFSM test.cpp
struct B_2_2
    : M::TrackedBase // injection!
{
    void substitute(Control& control,
                    Context&, const Time) const
    {
        if (entryCount() == 2) // from M::Tracked
            control.resume<A>();
    }
};
```

```cpp
template <...>
class Machine {

    struct Tracked {
        unsigned _entryCount = 0;

    };

};
```

```cpp
// from HFSM test.cpp
struct B_2_2
    : M::TrackedBase // injection!
{
    void substitute(Control& control,
                    Context&, const Time) const
    {
        if (entryCount() == 2) // from M::Tracked
            control.resume<A>();
    }
};
```

```cpp
template <...>
class Machine {

    struct Tracked {
        unsigned _entryCount = 0;

        // called by hfsm::Machine::BaseT<>
        // before hfsm::Machine::State::enter()
        inline void preEnter(Context&, const Time) {
            ++_entryCount;
        }





    };






};
```

```cpp
// from HFSM test.cpp
struct B_2_2
    : M::TrackedBase // injection!
{
    void substitute(Control& control,
                    Context&, const Time) const
    {
        if (entryCount() == 2) // from M::Tracked
            control.resume<A>();
    }
};
```

```cpp
template <...>
class Machine {

    struct Tracked {
        unsigned _entryCount = 0;

        // called by hfsm::Machine::BaseT<>
        // before hfsm::Machine::State::enter()
        inline void preEnter(Context&, const Time) {
            ++_entryCount;
        }

        // interface
        inline unsigned entryCount() const {
            return _entryCount;
        }
    };



};
```

```cpp
// from HFSM test.cpp
struct B_2_2
    : M::TrackedBase // injection!
{
    void substitute(Control& control,
                    Context&, const Time) const
    {
        if (entryCount() == 2) // from M::Tracked
            control.resume<A>();
    }
};
```

```cpp
template <...>
class Machine {

    struct Tracked {
        unsigned _entryCount = 0;

        // called by hfsm::Machine::BaseT<>
        // before hfsm::Machine::State::enter()
        inline void preEnter(Context&, const Time) {
            ++_entryCount;
        }

        // interface
        inline unsigned entryCount() const {
            return _entryCount;
        }
    };

    template <typename... TInjections>
    class BaseT;

    using TrackedBase = BaseT<Tracked>;
};
```

Advanced Library Interface :: State Injections

To add your own state injections:

```
// 1: inherit from hfsm::Machine::Bare
struct MyInjection
    : hfsm::Machine::Bare
{







};
```

Advanced Library Interface

To add your own state injections:

```
// 1: inherit from hfsm::Machine::Bare
struct MyInjection
    : hfsm::Machine::Bare
{
    // 2: implement any of:
    void preSubstitute(Context& _, const Time) const;
    void preEnter(Context& _, const Time);
    void preUpdate(Context& _, const Time);
    void preTransition(Context& _, const Time) const;
    void postLeave(Context& _, const Time);



};
```

Advanced Library Interface

To add your own state injections:

```
// 1: inherit from hfsm::Machine::Bare
struct MyInjection
    : hfsm::Machine::Bare
{
    // 2: implement any of:
    void preSubstitute(Context& _, const Time) const;
    void preEnter(Context& _, const Time);
    void preUpdate(Context& _, const Time);
    void preTransition(Context& _, const Time) const;
    void postLeave(Context& _, const Time);

    // 3: add interface:
    float getBlah() const;
};
```

Advanced Library Interface

To add your own state injections:

```
// 1: inherit from hfsm::Machine::Bare
struct MyInjection
    : hfsm::Machine::Bare
{
    // 2: implement any of:
    void preSubstitute(Context& _, const Time) const;
    void preEnter(Context& _, const Time);
    void preUpdate(Context& _, const Time);
    void preTransition(Context& _, const Time) const;
    void postLeave(Context& _, const Time);

    // 3: add interface:
    float getBlah() const;
};
```

```
// 4: inject it with hfsm::Machine::BaseT<>
struct MyState
    : hfsm::Machine::BaseT<MyInjection,
                           Machine::Tracked>
{

}
```

Advanced Library Interface

To add your own state injections:

```
// 1: inherit from hfsm::Machine::Bare
struct MyInjection
    : hfsm::Machine::Bare
{
    // 2: implement any of:
    void preSubstitute(Context& _, const Time) const;
    void preEnter(Context& _, const Time);
    void preUpdate(Context& _, const Time);
    void preTransition(Context& _, const Time) const;
    void postLeave(Context& _, const Time);

    // 3: add interface:
    float getBlah() const;
};
```

```
// 4: inject it with hfsm::Machine::BaseT<>
struct MyState
    : hfsm::Machine::BaseT<MyInjection,
                           Machine::Tracked>
{
    void update(Context& c, const Time t) {
        // 5: use:
        makeUseOf(getBlah());
    }
}
```

AGENDA

* INTRO
  * A Library for Video Games
  * High Level Design Overview

* SIMPLE STATE MACHINES
  * Example - Player Character
  * Library Interface for Simple State Machines
  * Example - Player Character, FSM

* HIERARCHICAL STATE MACHINES
  * Example - Complex Player Character
  * Library Interface for Hierarchical State Machines
  * Example - Complex Player Character, FSM
  * State Machine Complexity Intuition

* OUTRO
  * Advanced Library Interface
  * **Post-Mortem**
  * Future Work

* SUMMARY

Post-Mortem :: Good: Technical Investment Paid Off

Results:
* Better and more consistent code structure
* Less game object code
* Related logic is located more closely
* Explicit hierarchy made relationships between interacting features clear

Outcomes:
* Improved readability (fewest WTFs?/s ever)
* Fewer bugs, once the framework was de-bugged
* Adding new features to the existing game object logic was never easier

Post-Mortem :: Good: Unit Test!

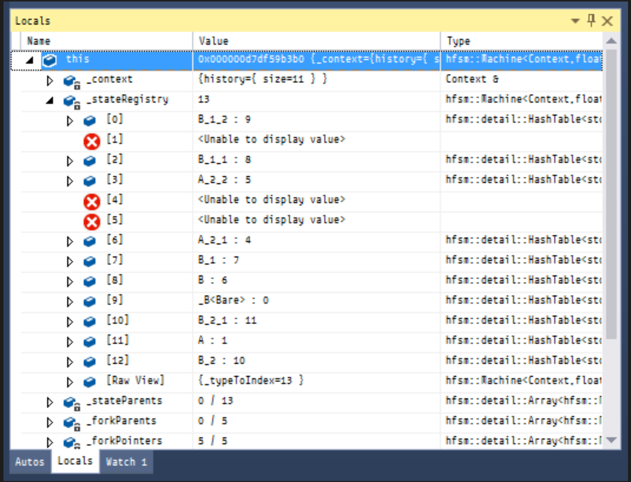
```
// Ensuring correctness in transition logic would be impossible without unit tests:

template <typename T>
struct HistoryBase {
    void preSubstitute(Context& _, const Time) const {
        _.history.push_back(Status{ Event::Substitute, hfsm::detail::TypeInfo::get<T>() });
    }
    void preEnter(Context& _, const Time) {
        _.history.push_back(Status{ Event::Enter, hfsm::detail::TypeInfo::get<T>() });
    }
    void preUpdate(Context& _, const Time) {
        _.history.push_back(Status{ Event::Update, hfsm::detail::TypeInfo::get<T>() });
    }
    void preTransition(Context& _, const Time) const {
        _.history.push_back(Status{ Event::Transition, hfsm::detail::TypeInfo::get<T>() });
    }
    void postLeave(Context& _, const Time) {
        _.history.push_back(Status{ Event::Leave, hfsm::detail::TypeInfo::get<T>() });
    }
};

// state A with HistoryBase<> injection
struct A : Machine::BaseT<HistoryBase<A>>;
```

Post-Mortem :: Good: Unit Test!

```
void main() {
    Context _;
    Machine::CompositeRoot<...> machine(_);

    machine.update(0.0f);
    const Status update1[] = {
        Status{ Event::Update, typeid(A) },
        Status{ Event::Transition, typeid(A) },
        Status{ Event::Update, typeid(A_1) },
        Status{ Event::Transition, typeid(A_1) },

        Status{ Event::Restart, typeid(A_2) },

        Status{ Event::Substitute, typeid(A_2) },
        Status{ Event::Substitute, typeid(A_2_1) },

        Status{ Event::Leave, typeid(A_1) },

        Status{ Event::Enter, typeid(A_2) },
        Status{ Event::Enter, typeid(A_2_1) },
    };
    for (unsigned i = 0; i < std::min(historySize, update1); ++i)
        assert(_.history[i] == update1[i]);
}
```

Post-Mortem :: Good: Native Debug Visualisation (.natvis)

E.g.: VS 2017 debugger view of hfsm::detail::HashTable<>:



Massively helpful when debugging!

Post-Mortem :: Bad: Long Template Names in Debug

```
Machine::CompositeRoot<
  Machine::Composite<A,
    Machine::State<A_1>,
    Machine::Composite<A_2,
      Machine::State<A_2_1>,
      Machine::State<A_2_2>
    >
  >,
  Machine::Orthogonal<B,
    Machine::Composite<B_1,
      Machine::State<B_1_1>,
      Machine::State<B_1_2>
    >,
    Machine::Composite<B_2,
      Machine::State<B_2_1>,
      Machine::State<B_2_2>
    >
  >
>
```

Post-Mortem :: Bad: Long Template Names in Debug

```
1>#hfsm\machine.inl(1281): error C2664:
'hfsm::Machine<Context,float,4>::_C<hfsm::Machine<Context,float,4>::_B<hfsm::Machine<Context,float,4>::Bare>,hfs
m::Machine<Context,float,4>::_C<A,hfsm::Machine<Context,float,4>::_S<A_1>,hfsm::Machine<Context,float,4>::_C<A_2
,hfsm::Machine<Context,float,4>::_S<A_2_1>,hfsm::Machine<Context,float,4>::_S<A_2_2>>>,hfsm::Machine<Context,flo
at,4>::_O<B,hfsm::Machine<Context,float,4>::_C<B_1,hfsm::Machine<Context,float,4>::_S<B_1_1>,hfsm::Machine<Con-
text,float,4>::_S<B_1_2>>,hfsm::Machine<Context,float,4>::_C<B_2,hfsm::Machine<Context,float,4>::_S<B_2_1>,hfsm:
:Machine<Context,float,4>::_S<B_2_2>>>>::_C(hfsm::Machine<Context,float,4>::_C<hfsm::Machine<Context,float,4>::_
B<hfsm::Machine<Context,float,4>::Bare>,hfsm::Machine<Context,float,4>::_C<A,hfsm::Machine<Context,float,4>::_S<
A_1>,hfsm::Machine<Context,float,4>::_C<A_2,hfsm::Machine<Context,float,4>::_S<A_2_1>,hfsm::Machine<Context,floa
t,4>::_S<A_2_2>>>,hfsm::Machine<Context,float,4>::_O<B,hfsm::Machine<Context,float,4>::_C<B_1,hfsm::Machine<Con-
text,float,4>::_S<B_1_1>,hfsm::Machine<Context,float,4>::_S<B_1_2>>,hfsm::Machine<Context,float,4>::_C<B_2,hfsm:
:Machine<Context,float,4>::_S<B_2_1>,hfsm::Machine<Context,float,4>::_S<B_2_2>>>> &&)':
cannot convert argument 1 from 'hfsm::detail::Array<T,5>' to 'const
hfsm::Machine<Context,float,4>::_C<hfsm::Machine<Context,float,4>::_B<hfsm::Machine<Context,float,4>::Bare>,hfsm
::Machine<Context,float,4>::_C<A,hfsm::Machine<Context,float,4>::_S<A_1>,hfsm::Machine<Context,float,4>::_C<A_2,
hfsm::Machine<Context,float,4>::_S<A_2_1>,hfsm::Machine<Context,float,4>::_S<A_2_2>>>,hfsm::Machine<Context,floa
t,4>::_O<B,hfsm::Machine<Context,float,4>::_C<B_1,hfsm::Machine<Context,float,4>::_S<B_1_1>,hfsm::Machine<Contex
t,float,4>::_S<B_1_2>>,hfsm::Machine<Context,float,4>::_C<B_2,hfsm::Machine<Context,float,4>::_S<B_2_1>,hfsm::Ma
chine<Context,float,4>::_S<B_2_2>>>> &'
```

Post-Mortem :: Bad: Long Template Names in Debug (Dodgy Workaround)

```
template <typename Tcontext>
class Machine {
private:                                                    public:




                                                            };
```

Post-Mortem :: Bad: Long Template Names in Debug (Dodgy Workaround)

```cpp
template <typename Tcontext>
class Machine {
private:
    template <typename...>
    class _B;

public:
    template <typename... TInjections>
    using BaseT = _B<TInjections...>;

};
```

Post-Mortem :: Bad: Long Template Names in Debug (Dodgy Workaround)

```
template <typename Tcontext>
class Machine {
private:
    template <typename...>
    class _B;

    template <typename T>
    class _S;

public:
    template <typename... TInjections>
    using BaseT = _B<TInjections...>;

    template <typename TClient>
    using State = _S<TClient>;

};
```

Post-Mortem :: Bad: Long Template Names in Debug (Dodgy Workaround)

```
template <typename Tcontext>
class Machine {
private:
    template <typename...>
    class _B;

    template <typename T>
    class _S;

    template <typename T, typename... TS>
    class _C;

public:
    template <typename... TInjections>
    using BaseT = _B<TInjections...>;

    template <typename TClient>
    using State = _S<TClient>;

    template <typename TClient, typename... TSubStates>
    using Composite = _C<TClient, TSubStates...>;

};
```

Post-Mortem :: Bad: Long Template Names in Debug (Dodgy Workaround)

```cpp
template <typename Tcontext>
class Machine {
private:
    template <typename...>
    class _B;

    template <typename T>
    class _S;

    template <typename T, typename... TS>
    class _C;

    template <typename T, typename... TS>
    class _O;

public:
    template <typename... TInjections>
    using BaseT = _B<TInjections...>;

    template <typename TClient>
    using State = _S<TClient>;

    template <typename TClient, typename... TSubStates>
    using Composite = _C<TClient, TSubStates...>;

    template <typename TClient, typename... TSubStates>
    using Orthogonal = _O<TClient, TSubStates...>;

};
```

Post-Mortem :: Bad: Long Template Names in Debug (Dodgy Workaround)

```cpp
template <typename Tcontext>
class Machine {
private:
    template <typename...>
    class _B;

    template <typename T>
    class _S;

    template <typename T, typename... TS>
    class _C;

    template <typename T, typename... TS>
    class _O;

    template <typename T>
    class _R;

public:
    template <typename... TInjections>
    using BaseT = _B<TInjections...>;

    template <typename TClient>
    using State = _S<TClient>;

    template <typename TClient, typename... TSubStates>
    using Composite = _C<TClient, TSubStates...>;

    template <typename TClient, typename... TSubStates>
    using Orthogonal = _O<TClient, TSubStates...>;

    template <typename TState>
    using Root = _R<TState>;
};
```

Post-Mortem :: Bad: Long Template Names in Debug (Dodgy Workaround)

A bit too many supplemental classes:

```
namespace hfsm::detail {















}
```

Post-Mortem :: Bad: Code Complexity

A bit too many supplemental classes:

```
namespace hfsm::detail {

    template <typename T>
    class ArrayView;







}
```

Post-Mortem :: Bad: Code Complexity

A bit too many supplemental classes:

```
namespace hfsm::detail {

    template <typename T>
    class ArrayView;

    template <typename T, unsigned TCapacity>
    class Array;    // : public ArrayView<T>

}
```

Post-Mortem :: Bad: Code Complexity

A bit too many supplemental classes:

```
namespace hfsm::detail {

    template <typename T>
    class ArrayView;

    template <typename T, unsigned TCapacity>
    class Array;    // : public ArrayView<T>

    template <typename TContainer>
    class Iterator; // for Array<>

}
```

Post-Mortem :: Bad: Code Complexity

A bit too many supplemental classes:

```
namespace hfsm::detail {

    template <typename T>
    class ArrayView;

    template <typename T, unsigned TCapacity>
    class Array;    // : public ArrayView<T>

    template <typename TContainer>
    class Iterator; // for Array<>

    template <typename T>
    class Wrap;     // for delayed construction of std::type_index




}
```

Post-Mortem :: Bad: Code Complexity

A bit too many supplemental classes:

```
namespace hfsm::detail {

    template <typename T>
    class ArrayView;

    template <typename T, unsigned TCapacity>
    class Array;    // : public ArrayView<T>

    template <typename TContainer>
    class Iterator; // for Array<>

    template <typename T>
    class Wrap;     // for delayed construction of std::type_index

    class TypeInfo; // : public Wrap<std::type_index>



}
```

Post-Mortem :: Bad: Code Complexity

A bit too many supplemental classes:

```
namespace hfsm::detail {

    template <typename T>
    class ArrayView;

    template <typename T, unsigned TCapacity>
    class Array;    // : public ArrayView<T>

    template <typename TContainer>
    class Iterator; // for Array<>

    template <typename T>
    class Wrap;     // for delayed construction of std::type_index

    class TypeInfo; // : public Wrap<std::type_index>

    template <typename TKey, typename TValue, unsigned TCapacity, typename THasher>
    class HashTable;
}
```

AGENDA

* INTRO
  * A Library for Video Games
  * High Level Design Overview

* SIMPLE STATE MACHINES
  * Example - Player Character
  * Library Interface for Simple State Machines
  * Example - Player Character, FSM

* HIERARCHICAL STATE MACHINES
  * Example - Complex Player Character
  * Library Interface for Hierarchical State Machines
  * Example - Complex Player Character, FSM
  * State Machine Complexity Intuition

* OUTRO
  * Advanced Library Interface
  * Post-Mortem
  * **Future Work**

* SUMMARY

Future Work :: Support User-Defined Time

'Time' is used for both duration and time_point, incompatible with <chrono>:

```
template <typename TContext = void, typename TTime = float, unsigned TMaxSubstitutions = 4>
class Machine {
    using Time = TTime;

    template <typename T>
    class _R final {
        update(const Time time);
    };

    struct Timed {
        inline auto entryTime() const { return _entryTime; }

        inline void preEnter(Context&, const Time time) { _entryTime = time; }

        Time _entryTime;
    };
};
```

Future Work :: Active State Chain Dump for Debugging

Dump currently active chain as plain text:

```
[Root]
  ├ State
  ├ CompositeRegion                 =>
  │  ├ State                        =>
  │  └ State                        =>
  └ [OrthogonalRegion]              =>
     ╟ [CompositeRegion]            =>
     ║  ├ State
     ║  └ [State]
     ╙ [State]
```

```
Root
 └ OrthogonalRegion
    ╟ CompositeRegion
    ║  └ State
    ╙ State
```

AGENDA

* INTRO
  * A Library for Video Games
  * High Level Design Overview

* SIMPLE STATE MACHINES
  * Example - Player Character
  * Library Interface for Simple State Machines
  * Example - Player Character, FSM

* HIERARCHICAL STATE MACHINES
  * Example - Complex Player Character
  * Library Interface for Hierarchical State Machines
  * Example - Complex Player Character, FSM
  * State Machine Complexity Intuition

* OUTRO
  * Advanced Library Interface
  * Post-Mortem
  * Future Work

* **SUMMARY**


Google
GIT


Google
GIT GUD

GIT GUD HUB

Features Business Explore Marketplace Pricing CAN HAZ hfsm Sign in or Sign up

Repositories 22 Code Commits Issues 20 Wikis 6 Users 4 Advanced search

22 repository results

Sort: **Best match**

andrew-gresyk/*HFSM* — C++ ★ 5

Hierarchical Finite State Machine Framework

Updated 28 days ago

permeakra/*hfsm* — Haskell ★ 1

Haskell finite state machines datatypes and baseops

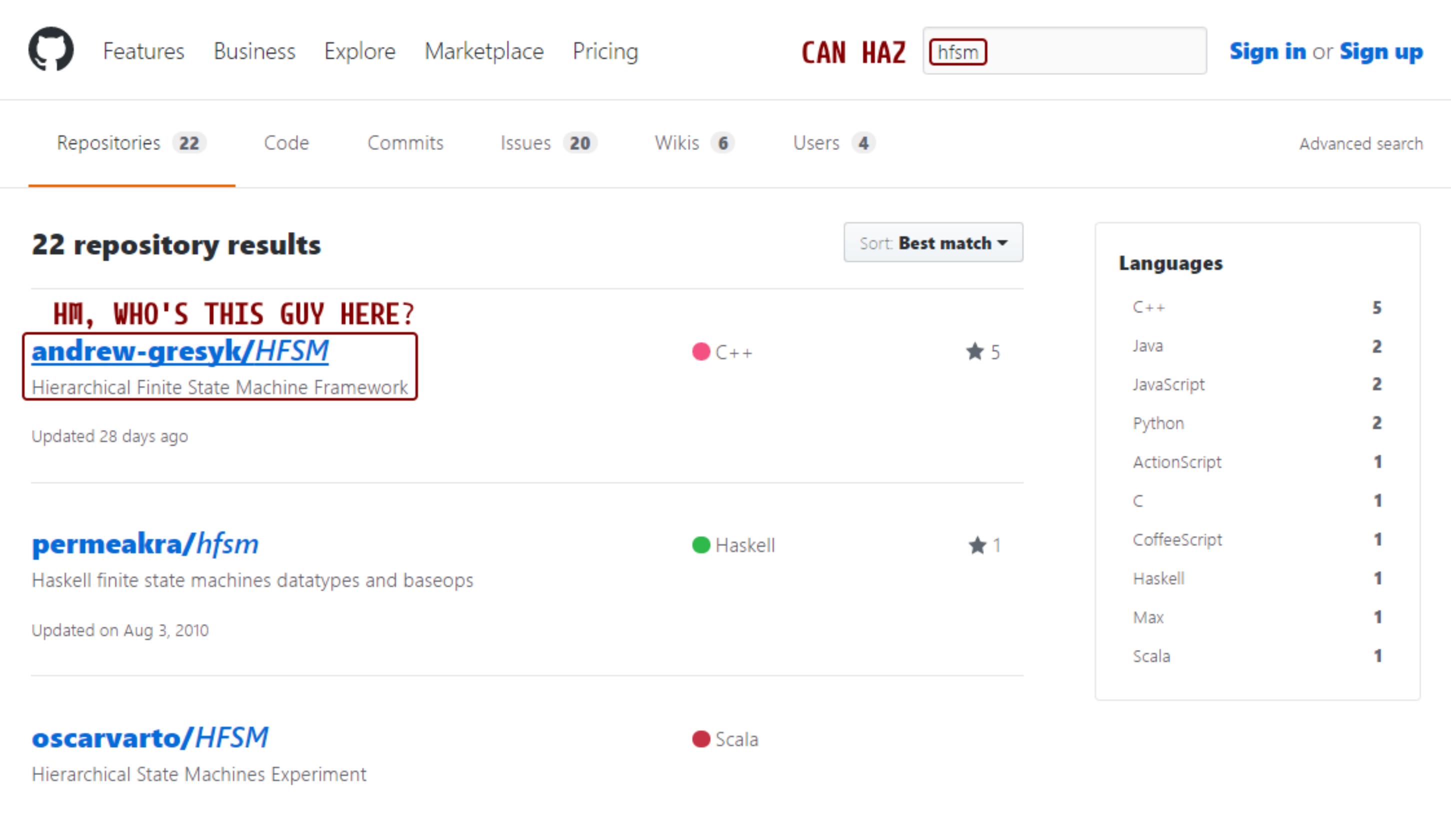
Updated on Aug 3, 2010

oscarvarto/*HFSM* — Scala

Hierarchical State Machines Experiment

Languages

C++	5
Java	2
JavaScript	2
Python	2
ActionScript	1
C	1
CoffeeScript	1
Haskell	1
Max	1
Scala	1


Features
Business
Explore
Marketplace
Pricing
CAN HAZ
hfsm
Sign in or Sign up
Repositories 22
Code
Commits
Issues 20
Wikis 6
Users 4
Advanced search
22 repository results
Sort: Best match
HM, WHO'S THIS GUY HERE?
andrew-gresyk/HFSM
Hierarchical Finite State Machine Framework
C++
5
Updated 28 days ago
permeakra/hfsm
Haskell finite state machines datatypes and baseops
Haskell
1
Updated on Aug 3, 2010
oscarvarto/HFSM
Hierarchical State Machines Experiment
Scala
Languages
C++ 5
Java 2
JavaScript 2
Python 2
ActionScript 1
C 1
CoffeeScript 1
Haskell 1
Max 1
Scala 1

Features Business Explore Marketplace Pricing

CAN HAZ hfsm

Sign in or Sign up

Repositories 22 | Code | Commits | Issues 20 | Wikis 6 | Users 4 | Advanced search

22 repository results

Sort: **Best match**

HM, WHO'S THIS GUY HERE?

C++ FOR LYFE!

andrew-gresyk/*HFSM* — C++ — ★ 5

Hierarchical Finite State Machine Framework

Updated 28 days ago

permeakra/*hfsm* — Haskell — ★ 1

Haskell finite state machines datatypes and baseops

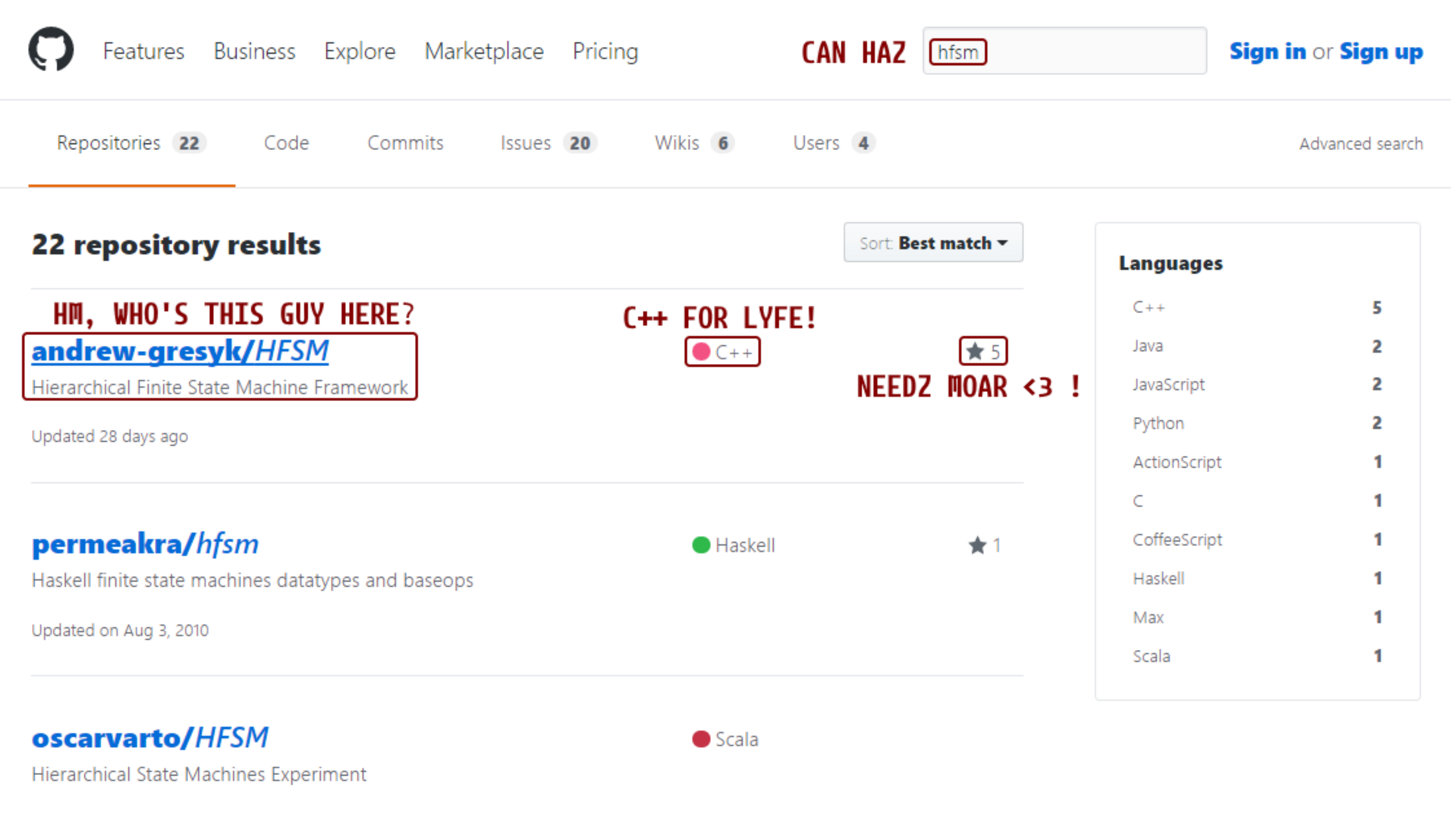
Updated on Aug 3, 2010

oscarvarto/*HFSM* — Scala

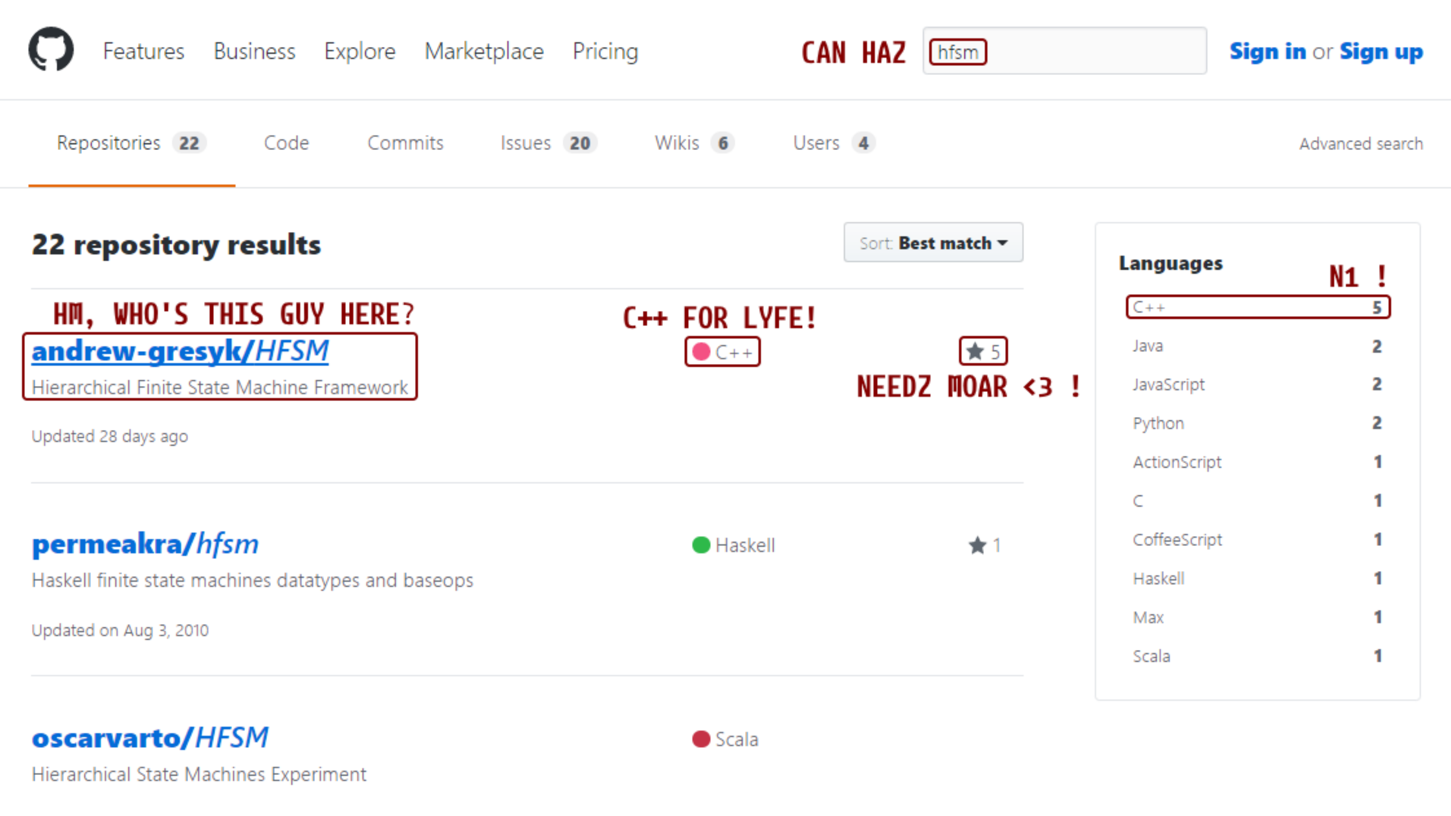
Hierarchical State Machines Experiment

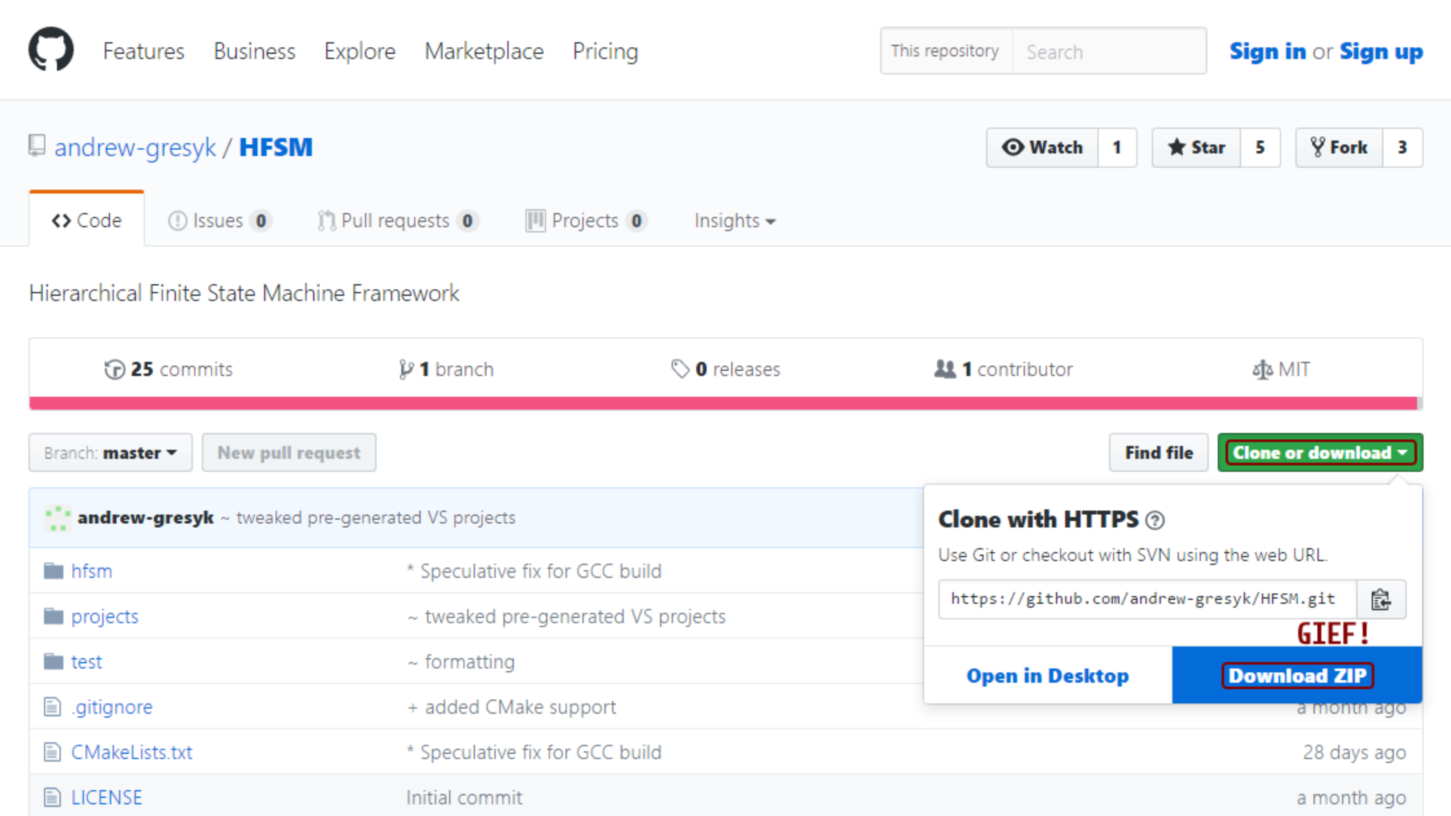
Languages

C++	5
Java	2
JavaScript	2
Python	2
ActionScript	1
C	1
CoffeeScript	1
Haskell	1
Max	1
Scala	1



Features
Business
Explore
Marketplace
Pricing
CAN HAZ
hfsm
Sign in or Sign up
Repositories 22
Code
Commits
Issues 20
Wikis 6
Users 4
Advanced search
22 repository results
Sort: Best match
HM, WHO'S THIS GUY HERE?
C++ FOR LYFE!
andrew-gresyk/HFSM
C++
5
Hierarchical Finite State Machine Framework
NEEDZ MOAR <3 !
Updated 28 days ago
permeakra/hfsm
Haskell
1
Haskell finite state machines datatypes and baseops
Updated on Aug 3, 2010
oscarvarto/HFSM
Scala
Hierarchical State Machines Experiment
Languages
C++ 5
Java 2
JavaScript 2
Python 2
ActionScript 1
C 1
CoffeeScript 1
Haskell 1
Max 1
Scala 1


Features
Business
Explore
Marketplace
Pricing
CAN HAZ
hfsm
Sign in or Sign up
Repositories 22
Code
Commits
Issues 20
Wikis 6
Users 4
Advanced search
22 repository results
Sort: Best match
HM, WHO'S THIS GUY HERE?
andrew-gresyk/HFSM
Hierarchical Finite State Machine Framework
C++ FOR LYFE!
C++
5
NEEDZ MOAR <3 !
Updated 28 days ago
permeakra/hfsm
Haskell
1
Haskell finite state machines datatypes and baseops
Updated on Aug 3, 2010
oscarvarto/HFSM
Scala
Hierarchical State Machines Experiment
Languages
N1 !
C++ 5
Java 2
JavaScript 2
Python 2
ActionScript 1
C 1
CoffeeScript 1
Haskell 1
Max 1
Scala 1


Features
Business
Explore
Marketplace
Pricing
This repository
Search
Sign in or Sign up
andrew-gresyk / HFSM
Watch 1
Star 5
Fork 3
Code
Issues 0
Pull requests 0
Projects 0
Insights
Hierarchical Finite State Machine Framework
25 commits
1 branch
0 releases
1 contributor
MIT
Branch: master
New pull request
Find file
Clone or download
andrew-gresyk ~ tweaked pre-generated VS projects
Clone with HTTPS
Use Git or checkout with SVN using the web URL.
https://github.com/andrew-gresyk/HFSM.git
GIEF!
Open in Desktop
Download ZIP
hfsm
* Speculative fix for GCC build
projects
~ tweaked pre-generated VS projects
test
~ formatting
.gitignore
+ added CMake support
a month ago
CMakeLists.txt
* Speculative fix for GCC build
28 days ago
LICENSE
Initial commit
a month ago

README.md

HFSM (Hierarchical Finite State Machine) Framework

Header-only heriarchical FSM framework in C++14, completely static (no dynamic allocations), built with variadic templates.

Compiler Support

- Visual Studio 2015+
- Visual Studio 2017 with Clang codegen v2
- GCC 6.3.1
- Clang 3.9.1

Basic Usage

```
// 1. Include HFSM header:
#include <hfsm/machine.hpp>

// 2. Define interface class between the FSM and its owner
//    (also ok to use the owner object itself):
struct Context { /* ... */ };
```

Basic Usage

```
// 1. Include HFSM header:
#include <hfsm/machine.hpp>

// 2. Define interface class between the FSM and its owner
//    (also ok to use the owner object itself):
struct Context { /* ... */ };

// 3. (Optional) Typedef hfsm::Machine for convenience:
using M = hfsm::Machine<OwnerClass>;

// 4. Define states:
struct MyState1 : M::Bare {

   // 5. Override some of the following state functions:
   void substitute(Control& c, Context& _, const Time t);
   void enter(Context& _, const Time t);
   void update(Context& _, const Time t);
   void transition(Control& c, Context& _, const Time t);
   void leave(Context& _, const Time t);
};

// 6. Declare state machine structure:
using MyFSM = M::CompositeRoot<
   M::State<MyState1>,
   M::State<MyState2>,
```