The Troops have 5 states, with the capability to add more later. These are Roaming, to Ganon, Held by Ganon, Returning to Combat, and Special Attack. (To Ganon and held by Ganon could potentially be a part of one of the other states.) In each state the troops behave differently, and have different animations / no animations. So, here’s a breakdown of what one of these state machines could look like and all of the parts needed for it.

//All units need to inherit from a common base Object

class Object

{

//All parameters, which all units have.

//Methods all objects need.

virtual void Update();

virtual void HandleMessage(Telegram telegram);

}

//This is the base state class.

//I need to find the keyword for making a template in C#

template<class Object>

class State

{

//Methods so states can have different behaviours depending on what they’re doing.

virtual void Enter(Object);

//State runs this when no state changes are occuring

virtual void Execute(Object);

virtual void Exit(Object);

//Tells state what to do when receiving a message

virtual void OnMessage(Object, Telegram);

}

//This is what the base state machine looks like

//I have to find the keyword for making a template in C#.

template<class Object>

class StateMachine

{

State<Object> currentState;

State<Object> previousState;

State<Object> globalState;

Object owner;

StateMachine(Object owner)

{

//pass all parameters their default values here.

}

//Might need to change for unity

void Update()

{

if(globalState)

globalState.Execute(owner);

if(currentState)

currentState.Execute(owner);

}

//Telegram is a wrapper class that holds all information other

//Units might need.

void HandleMessage(Telegram telegram)

{

currentState.OnMessage(owner, telegram);

if(globalState)

globalState.Onmessage(owner, telegram);

}

void ChangeState(State<Object> newState)

{

previousState = currentState;

currentState.Exit(owner);

currentState = newState;

currentState.Enter(owner);

}

}

//An example of the Moblin class

Class Moblin : Object

{

private StateMachine<Ball> stateMachine;

virtual void Update();

virtual void HandleMessage(Telegram telegram);

}

//An example of the Moblin StateMachine

//We could probably break the inheritence down more so each class inherits a different type of //state machine for each type of action.

//For instance

Moblin<Attack> : State<Moblin>

{

//class stuff goes here

}

//But for now this will be a simpler way to do it

Class MoblinAttack : State<Moblin>

{

//We’re making a class that guarantees only one can exist.

Private static MoblinAttack moblinAttack

private static MoblinAttack()

{

moblinAttack = new MoblinAttack();

}

public static MoblinAttack Instance()

{

if(moblinAttack == null)

//Create new moblinAttack

return moblinAttack

}

public void Execute(Moblin mob)

{

//Do moblin attack stuff

//changes state based on condition

//Can be in any of the public methods except for the constructor

//and Instance

if(true)

//Change state.

}

public void Enter(Moblin mob)

{

//When entering the state do this

}

public void Exit(Moblin mob)

{

//What happens when the state is exited.

}

public void OnMessage(Moblin mob, Telegram telegram)

{

//What happens when a message is received

}

}

//How messaging could work

//Might work as a singleton (When you can have only one of an object.)

struct Messenger

{

list bankOfObjects = new list();

list bankOfTelegrams = new list();

//Constructor omitted, but it will probably be like MoblinAttack’s

public AddTo(Object object, Telegram telegram)

{

bankOfObjects.Add(object);

bankOfTelegrams.Add(telegram);

}

public void Send()

{

for(int = 0; i < bankOfObjects.size(); i++)

{

for(int j = 0; j < bankOfTelegrams.size(); j++)

{

bankOfObjects.at(i).HandleMessage(bankOfTelegrams.at(j));

}

}

}

}

//The messanger could change severely during break.

//ideally we’ll have one class that holds all of the moblins and their corresponding statemachines

//that is responsible for ensuring each receives the correct message, but that takes longer to //make.