Member Login
Email:
Password:
Login
Forgot Password? Sign Up

PROGRAMMING
ART
AUDIO
DESIGN
PRODUCTION
BIZ/MARKETING

Latest Jobs
View All RSS
August 6, 2017
- Disruptor Beam
  Sr. QA Engineer
- Disruptor Beam
  DevOps Engineer
- 2K
  SENIOR SERVER ENGINEER
- OtherSide Entertainment
  Senior Gameplay Engineer
- Respawn Entertainment
  Senior Software Engineer
- Sony PlayStation
  Senior Animation Programmer

Latest Blogs
View All Post RSS
August 6, 2017
- Persona 5, Cartoon Cats, Depthless Evil, and Dating Your Teacher. [7]
- Cohesive Game Experience, Part I
- Kickstarter and Games – 2017 mid-year status update [2]
- Game Producer Voodoo [7]
- Pricing games is hard! [14]

Press Releases
August 6, 2017
Games Press
View All RSS

About
- Editor-In-Chief:
  Kris Graft
- Editor:
  Alex Wawro
- Assignment Editor:
  Chris Baker
- Contributors:
  Chris Kerr
  Alissa McAloon
  Emma Kidwell

Features

Visual Finite State Machine AI Systems

by Sunbir Gill [Programming, Game Developer Magazine]

Post A Comment Share G+

November 18, 2004

Finite State Machines (FSMs) are a simple and efficient method to implement many game features. FSMs are an element of, or can solve in entirety, many of the problems encountered in game programming: AI, input handling, player handling, UI, and progression. They may be diagrammed using a standard diagram format called a directed graph, which is easy to read and understand, even for non-programmers. It is a simple process to convert directed graphs to state tables and vice-versa. It is also easy to optimize an FSM to a minimum set of states.

This article will examine an approach to implementing FSMs such that no programming is necessary to take an FSM diagram from design-time to executing it in-game.

The usefulness of this system is that it saves programmer time. It offsets the work of creating FSMs to a designer. The best areas to use this system are where the savings can be leveraged the most. A repetitious programming task such as state-machine AI is a prime candidate.

A visual state machine system is a solution to designer-created AI problems that I've had in the past. Initially, we used a sequential script system to create AI on our projects. The system required scripting-knowledgeable designers and the AI scripts relied heavily on messaging (partially to compensate for lack of state tracking) which had a negative impact on performance. Our scripted AI ended up being composed of several different scripts. The scripted AI mismanaged its projectiles and spawned child entities that later led to memory and performance issues at the end of the project.

The technique described here was conceived to solve our problems with creating AI for handheld games. State-machines are a simple, effective, and efficient way to model AI on resource-limited platforms, and shorter development schedules make it beneficial to offset the work of creating AI and other game-world entities to designers.

Working the System

The elements of our system begin with Microsoft Visio, which is the front-end GUI that will be used to layout state machine diagrams. The FSM stencil is a document maintained by the programmer and created to define the shapes which designers will be using in their diagrams. Once the FSM has been laid out by a designer, it will be saved as an XML document. This is the final build asset for the game.

The real work for the programmers are the tools used to parse and interpret the XML FSM drawing. The first tool is a command-line tool that converts an XML FSM drawing into state table data. Next comes the XML parser, which does most of the command-line tool's work. Finally, the in-game FSM execution class is the code that executes the FSM table data at run-time.

Reference documents and source code described here can be obtained from www.gdmag.com.

The sample AI System presented here describes the data and logic required to implement an AI character that walks back and forth along the x axis, similar to a "Goomba" (commonly referred to as a patroller) in *Super Mario Bros*.

Visio comes with dozens of templates to create accurate diagrams for many different standards. Users can also create templates, called stencils. Stencils can define the look and attributes of shapes and connectors that together are used to compose diagrams. It makes an ideal user interface for an FSM system and eliminates the need for any custom GUI tools.

Drawings and stencils can be saved or converted to a variety of formats. In this system we maintain stencils and drawings in XML format (*.vsx and *.vdx, respectively).

Shapes

In this sample AI System implementation, we will define 3 shapes: the state, the initial state, and the transition.

Of noticeable absence is a final state. We handle final states in the system implicitly. Any transition that ends with no state is considered to have entered a final state. Once an AI entity enters a final state it will be removed from the game.

Custom Properties

Each shape can have a set of user modifiable attributes called custom properties. The custom property feature supports several types: strings, integers, booleans, drop-lists, and more. With the exception of the name attribute (a string), we exclusively use drop-lists. The reasoning behind this will be seen when we discuss how the game code maps these attributes to data. For our Goomba system we've defined the custom properties listed in Table 1.

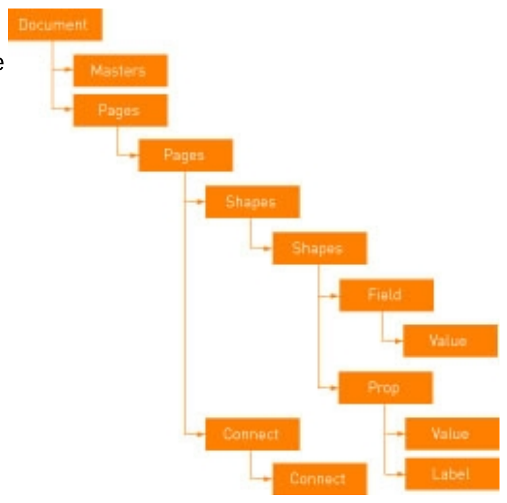
Figure 1 is a screenshot of our Goomba AI in Visio. The patroller walks back and forth between two trigger entities. When it gets hit by the player, it is defeated and enters the final state. The Goomba will hurt the player if they touch it (described by the HitType property; a value that lets the player know how to handle the AI collision data).

XML for Microsoft Visio Schema

Once a Visio stencil and a FSM diagram for the AI have been created, the next stage is to convert the relevant diagram data into a state table. The table should be a 2D array that is an equivalent model of the state machine and is suitable for processing in code.

The Visio XML diagram format is deep and complex. Fortunately, for our purpose of converting the XML to data, we need only concern ourselves with the sub-tree of the structure outlined in Figure 2 and described in Table 2.

By iterating through these tags the tool can obtain all the information required to convert a state-diagram into a state table.



A sub-tree of a Visio drawing XML tag hierarchy.

Parsing XML

There are many freely available XML parsing libraries. In our example, we will use Microsoft .Net's C++ library XMLDataDocument class. It acts as a DOM parser that both loads and contains the document. Accessor member functions make the iteration of nodes easy. DOM parsers are not as fast as single-pass SAX parsers, so if your project needs to support many AI constructs, and build-time is a priority, you may want to consider other options.

Table 1: State and Initial State	
Namep	user-defined string
Animation	drop-list with the following options:
	• existing • moveRight • moveLeft • death
XMovement	drop-list with the following options:
	• existing • right • left • death
Transition	
Condition	drop-list with the following options:
	• hitTriggerLeft • animationDone • hitTriggerRight • hitByPlayer

Table 2: Definitions for Figure 2	
... _Masters	This sub-tree contains stencil object information that will be required when generating output data. This information can be very useful in catching human error, i.e. if the incorrect stencil was being used. It can also be used to automatically generate the state definition header file that we present in our sample code.
... _Shape	An instance of a Stencil object. Could either be a state or transition. Has a unique integer attribute named ID.
... _Shape_Field_Value	This tag holds the name of the state or transition.
... _Shape_Prop	This tag holds a property of the state or transition.
... _Shape_Prop_Label	This tag holds the property name.
... _Shape_Prop_Value	This tag holds the user-selected custom property value.

... _Connect	This tag holds a transition connection from state to state. It has the following attributes: • FromSheet-The ID of the transition shape. • ToSheet-The ID of the state. • FromCell-A value of either "BeginX" or "EndX" representing the start and end points of the connector line respectively.

Parsing the diagram XML based on the aforementioned subset of the Visio schema should be straightforward. The sample code in Listing 1 illustrates how you can cleanly iterate through relevant nodes.

Listing 1 is a function that iterates through the Master tags in a document by their unique ID attribute. If the index is out-of-range, a NULL pointer is returned. Iterator functions are useful for simplifying the output code generation logic.

```
static XmlNode* getMaster(XmlDataDocument* xmlDoc, int ID)
{
XmlNode* pResult = NULL;
XmlNodeList* pElemList = xmlDoc->GetElementsByTagName("Master");
for (int i = 0; i < pElemList->Count; i++)
{
XmlAttributeCollection* pAttributes = pElemList->Item(i)->get_Attributes();
XmlAttribute* pAttrib = dynamic_cast(pAttributes->GetNamedItem("ID"));
if (Int32::Parse(pAttrib->Value) == ID)
{
pResult = pElemList->Item(i);
break;
}
}
return pResult;
}
```

Listing 1. A tag iterator function.

The Tool

The game engine will require a command-line exporter tool to convert the XML data to FSM data tables in C++ source, or a binary data file, as part of the build process. By using an XML parser to store the diagram data, we can iterate through the tags described in the previous section and collect the information we need. In Listing 2, we will generate C++ source code that can be compiled and linked into the binary executable.

```
namespace AI
{
// transitions
enum TransitionTestType
{
TEST_none = -1,
TEST_hitTriggerLeft = 0,
TEST_hitTriggerRight,
TEST_animationDone,
TEST_hitByPlayer,
};
struct TransitionTest
{
TransitionTestType mType;
int mValue;
};
// states
enum StateEntryActionType
{
ACTION_setAnimation,
ACTION_setXMovement,
};
enum Animation
{
ANIMATION_moveLeft,
ANIMATION_moveRight,
ANIMATION_death,
NUM_ANIMS,
};
enum XMovement
{
XMOVEMENT_left,
XMOVEMENT_right,
XMOVEMENT_death,
NUM_MOVEMENT,
};
struct StateEntryAction
{
StateEntryActionType mType;
int mValue;
};
struct State
{
const int* mTransitions;
int mNumTransitions;
const StateEntryAction* mActions;
int mNumActions;
};
// param
struct Param
{
const TransitionTest* tests;
int numTests;
const State* states;
int numStates;
const int* stateTable;
};
}
```

Listing 2. "ai.h" - Header file defining data array indices and structures referenced by generated output

Some useful data structure definitions shown in Listing 2 will be used by our generated output. Listing 3 is the actual generated output created by our command-line tool. In Listing 3 we have hand-rolled the file, but a mature tool could generate it directly from the stencil XML.The following list summarizes the purpose of the definitions used in Listing 2:

- The TransitionTestType enumeration names are mapped one-to-one with the custom properties drop-list for the Visio stencil transition shape.
- The TransitionTest structure is used to store custom properties from the transition shape.
- The StateEntryAction enumeration is a list of unique IDs for every action we could perform upon entry of state. In our example, we will only be changing movement along the x axis and/or playing a different AI animation.
- The Animation and XMovement enumerations are mapped one-to-one with custom property drop-lists for the state shape.
- The StateEntryAction structure describes an action to take upon entry of a state. The mValue member holds the index into the data array of the type specified by the mType member.
- The State structure is composed of two arrays: an array of possible transitions to other states and an array of actions to take upon state entry. The mTransitions array is a list of indices into an array of TransitionTest structures. This indirection allows us to compress transition data by avoiding duplicate structures.
- Everything is tied together in the Param struct. It contains arrays of TransitionTest and State structures. The stateTable array is an *n*-by-*m* 2D array where n is the number of states and m is the number of transitions. To find the destination state from state q upon successful transition test t is simply a matter of accessing index (t * numStates + q) in stateTable.

```
#include "_ai_goomba.h"
using namespace AI;
// transitions
const int numTransitionTests = 5;
const TransitionTest transitionTests[numTransitionTests] =
{
{ TEST_hitTriggerRight, 0 },
{ TEST_hitTriggerLeft, 0 },
{ TEST_hitByPlayer, 0 },
{ TEST_animationDone, 0 },
};
// states
const int numStates = 3;
// state 0
const int numState0Transitions = 2;
const int state0Transitions[numState0Transitions] = { 0, 2, };
const int numState0Actions = 2;
const StateEntryAction state0Actions[numState0Actions] =
{
{ ACTION_setXMovement, XMOVEMENT_left },
{ ACTION_setAnimation, ANIMATION_moveLeft },
};
// state 1
const int numState1Transitions = 2;
const int state1Transitions[numState1Transitions] = { 1, 2, };
const int numState1Actions = 2;
const StateEntryAction state1Actions[numState1Actions] =
{
{ ACTION_setXMovement, XMOVEMENT_right },
{ ACTION_setAnimation, ANIMATION_moveRight },
};
// state 2
const int numState2Transitions = 1;
const int state2Transitions[numState2Transitions] = { 3, };
const int numState2Actions = 2;
const StateEntryAction state2Actions[numState2Actions] =
{
{ ACTION_setXMovement, XMOVEMENT_death },
{ ACTION_setAnimation, ANIMATION_death },
};
const State states[numStates] =
{
// state 0 - start state
{
// transitions
state0Transitions,
numState0Transitions,
// actions
state0Actions,
numState0Actions,
},
// state 1
{
// transitions
state1Transitions,
numState1Transitions,
// actions
state1Actions,
numState1Actions,
},
// state 2
{
// transitions
state2Transitions,
numState2Transitions,
// actions
state2Actions,
numState2Actions,
},
};
const int stateTable[numStates * numTransitionTests] =
{
1, 0, 2, 0,
1, 0, 2, 1,
2, 2, 2, -1,
};
const Param ai_ref::param =
{
transitionTests,
numTransitionTests,
states,
numStates,
stateTable,
};
```

Listing 3. "ai_goomba.cpp"—The generated output data file.

Run-time

Listing 4 describes a simple execution engine for our Goomba state machine. Additional sample code may be downloaded from www.gdmag.com. The tool output leads to a very data-driven, lightweight, and efficient implementation of FSM execution code. We use an instance of an FSM executor class, AIObject, a descendant of a game entity class, VisibleGameObject, to run the FSM in the game.

The update() member function handles state machine execution. It is executed as an entity base class override that is called once per frame. It tests all outgoing transition tests for the current state and transitions to a new state if a test is successful. If entering a final state, designated by a -1 index in the state table, we make a base class call setDeletePending() to let the entity manager know that this entity should be removed from the game world.

The receive() member function processes asynchronous messages describing entity-entity interactions. We resort to using flags here to handle transition tests for these cases. After every message the state machine is updated with update() in addition to the regular per-frame call.

The enterState() member function handles the transitioning from one state to another. All state-entry actions are performed here. The value members of entry actions are actual indices into data arrays.

As seen by the sample code, this system introduces a very streamlined programming model. Implementation is largely a set of the VisioStencil-mapped, granular transition-test cases and state-entry routines. Execution is completely handled within the update() member function and this encapsulation makes for easy tracing and debugging.

```
                class AIObject : public VisibleGameObject
{
public:
AIObject(
const AI::Param& param,
const Animation** anims,
const MoveParam2D* moves,
const Position& worldLocation
);
virtual ~AIObject();
bool receive(const Hit& hit);
void update();
protected:
void enterState(const AI::State& state);
void setAnimation(AI::Animation anim);
void setMovement(AI::XMovement move);
const AI::Param& mParam;
const Animation** mAnims;
const MoveParam2D* mMoves;
int mCurrentStateIndex;
union
{
unsigned int mMask;
bool mHitByPlayer;
bool mHitTriggerLeft;
bool mHitTriggerRight;
} mFlags;
};
```

Listing 4. "AIObject.h"—Our AI Entity class that executes upon generated FSM output data.

Summary

Run-time performance is about as good as you could expect from a hand-rolled state machine. Potential space and time problems can be introduced during design. A poorly designed state machine can be confusing, require more states and transitions, and execute more code than necessary, but this is true for any such system. This problem can be partially solved by the tool, since FSMs are easily optimized.

The primary limitation of this system is that it does not scale well. As the number of states and transitions increase, the complexity of the document increases as well. Simply put, it is difficult to arrange large state-machine diagrams so that they are readable and understandable.

This FSM system can be expanded upon in several ways. Additional functionality can be achieved by supporting new shapes and new shape properties. Handling messages with a message-handler shape is one way to support asynchronous messaging without having to add transitions to every state. Integrating a stack can extend this solution as well. It would essentially make it a pushdown automata that is capable of solving a larger set of problems. Stack operations could be considered as just another type of state-entry action.

With XML, the future of game tools development is poised to make a radical shift. As the format is supported by more existing software packages, creation of large custom-GUI editors, application plug-ins, and scripts will become a thing of the past. XML data conversion, the driving idea behind this system, builds upon and extends functionality of current software and thus requires less training, less engineering, less support, and as a result, is a far more cost-effective tool solution.

[View All Jobs]

Top Stories

Building replayability into the macabre gameplay of *Kindergarten*

Is it worth it to make DLC? Indies share their DLC attach rates

Steam attracts an average of 1.5M new customers a month

VRDC Speaker Q&A: John Austin on building MR experiences in the real world

Comments

Login to Comment