A Question of Character: Rules to Play by

Ian Lane Davis, Technical Director

Activision, Inc.
3100 Ocean Park Blvd, Santa Monica, CA 90405
idavis@activision.com, akiam@alum.dartmouth.org

Abstract

Almost every type of computer game has some sort of AI scripting language which enables level designers to script a story arc for a mission or game level and to provide personalities and individual responses to different characters in a game. And whether the "characters" in a game are 3D polygonal virtual actors the player explicitly interacts with or virtual generals who direct massive armies but are never themselves seen, the goal of the AI is to make the characters come to life. Many scripting languages in games have some serious problems in speed or ease of use. Thus, we created a character management system we call the CCA (Character Control Architecture) and an efficient and easy to use rules system we call the IIIE (Ian's Improved Inference Engine) for an unshipped 3D action/adventure title, Planetfall 2: The Other Side of Floyd. The system was designed for maximal extensibility and flexibility, and currently we are using the CCA & IIIE for controlling the mission scripting and tactical ship AI for a 3D Real-Time Strategy game (RTS)

Introduction

Planetfall 2: The Other Side of Floyd was a 3D comical Action/Adventure game focussed around a hapless space traveler who had the misfortune to hook up with two dysfunctional robots named Floyd and Oliver. The original games that these characters came from (*Planetfall* and *Stationfall*) were text-only games and the transition to a real-time 3D graphical game created AI challenges such as path planning in a 3D environment, inverse kinematics systems for motion planning, and strategic AI for the player's foes. However, the biggest AI challenge was to create the logic and interaction with Floyd & Oliver, AI buddies who would be both part of the story and tools for the player. These robots were to follow the player around and perform a wide variety of tasks such as opening doors, fetching items, delaying enemies, and causing havoc. Towards this end, we created the CCA (Character Control Architecture) and the IIIE (Ian's Improved Inference Engine).

Philosophy

Two principle elements involved in bringing the characters to life are storytelling and emergent behaviors. Storytelling is scripted glimpses at the characters' personalities. Emergent behaviors are actions or combinations of actions performed by a character or characters that come from a combination of the characters' basic traits and abilities and the potentially unpredictable opportunities and situations that can arise. Storytelling gives the level designer control over the basic perception of the characters by the player, while emergent (or opportunistic) behavior allows the player to feel that she is experiencing and influencing a unique game experience. Balancing the storytelling and emergent behaviors is the trick to making an immersive game like Planetfall succeed in presenting believable and distinct characters, and in making an RTS mission seem both unique and responsive to a player's strategies.

Overview

In our CCA, each character is treated as an independent agent. Each character maintains a current plan of action, state, and memory for itself, and when the situation demands that a new decision be made, the character calls the inference engine to determine a new course of action.

There are five principle elements: The CCA (Character Control Architecture), the IIIE rules, the Action Scripts, the Atomic Actions, and the Character Attributes. The CCA and the Atomic Actions are general algorithms (coded in C++) common to all characters, while the IIIE rules, Action Scripts, and Character Attributes are specific to each character and created with a designer-friendly grammar.

The behavior of a character is determined by the different elements of the Behavior Pyramid shown below:

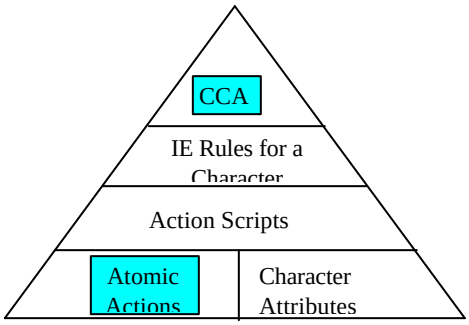

Figure 1. The Behavior Pyramid for a character. The CCA and Atomic Actions are part of the executable, while the rest is configurable by the level designer.

- CCA: The Character Control Architecture controls how a character manages its current plan(s) and when it calls the Inference Engine to determine a new course of action. The sophistication of the CCA is that it handles interruptions of plans elegantly, monitors changes in the character's environment, and could allow the resumption of interrupted plans. Every character in the game is instantiated as a CCA entity, including a "MISSION" entity that performs logic and actions for the environment.

- IIIE Rules: Each character has its own set of rules that fully define which actions it can initiate under any given circumstances. This makes the search for a specific character's next plan/action efficient (since the character's rules are separate from those of other characters), and it allows us to choose a new action for a specific character on demand. The "left-hand sides" of the rules are conditionals based on the Game & Character Attributes. The "right-hand sides" are tokens for specific Action Scripts. IIIE rules are loaded and unloaded in sets (by file), and a given character can have several different sets loaded at any given time (such as local and global rules). Sets of rules are defined either for a particular character or for a group of characters, and each character may belong to as many groups as the designer specifies (and the character inherits all rules sets attached to those groups).

- Action Scripts: Each action script is a description of how a specific plan or action looks, sounds, and plays out. It can contain motion instructions, sounds, & animations. It can set or unset character attributes. It specifies which Atomic Actions are part of a plan for the CCA to execute. The plan consists of a series of sequential steps, each of which can contain several simultaneous atomic actions, and it can potentially have steps that are dependent on the successful execution of previous steps (the latter is not implemented in the current version). As with IIIE rules, Action Scripts (AS) are loaded and unloaded in sets (by file), and several different sets can be loaded at any given time (such as local and global actions). A specific AS is referred to in the rules by a string identifier, such as Launch_Starcruiser_Attack or Hit_Lloyd. Action Scripts may be used by several characters

- Character Attributes: Everything we need to know about a character for the given environment. There are two types of attributes: permanent and situational. Permanent attributes are defined by the particular game: they could include location, orientation, current plan, etc. Situational attributes are defined by the designer in the action script files. Examples might be simple booleans like HAS_ID_CARD_IN_HAND. They could also be integers or floating point, like TIMES_FLOYD_HAS_HIT_ME or DIST_2_STAR. The situational attributes are loaded along with the Action Scripts and IIIE rules. The most interesting attributes are "hot variables" which are described later with the IIIE. Some attributes act as global variables (such as a user defined variable OPEN_AIRLOCK), and some have an instance attached to each character (such as Floyd.HITPOINTS).

- Atomic Actions: The very fundamental building blocks of the characters' behaviors are the Atomic Actions. These are just functions in the main game core that handle basic motions, sounds, & other interactions with the environment. Examples might be *plan_motion* (to_vector), *follow_character*(whom_to_follow), or *attack_enemy*(ship_number).

The Control Loop

A major part of the balance comes from designing the control loop of the game so that each character gets a fair chance to react to changes in the environment in a timely manner. The control loop must act like a process scheduler in an operating system. Each character needs to get its fair share of processing time, the control loop must not waste time on characters that don't need processing, and the whole thing has to happen efficiently. The model we are using is the "round robin" model [Tanenbaum87], in which there is a list of characters, and at each pass through the control loop, each character is given a finite chunk of time in which to perform some AI.

At the very highest level of control, the main loop of the game looks like this:

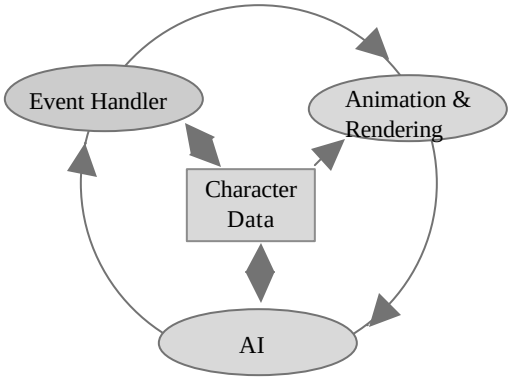

Figure 2. The Main Loop of Planetfall 2

The three principle components are the Event Handler, the Animation & Rendering Module, and the AI. The Event Handler is responsible for all input from the character as well as system events. The Animation & Rendering Module keeps the screen fresh with the latest projection of the game world. Finally, the AI determines and executes the characters' actions. In any given cycle of the game, each

module is allocated some amount of time (and, correspondingly, some amount of CPU processing) to perform its tasks. The data for each character is stored in one central repository. All of the relevant information about a character's state, including position and joint angles, is stored in a central, easy-to-access data structure.

There are two main components to the AI Loop: setting plans *for all characters* and executing plans *for all characters*. The main reason the planning and execution are separate is so that each character works with the same information (execution may change positions, joint angles, or other attributes). Also, the disjunction between planning and simulating allows us to update the simulation and drawing more frequently than we need to plan. During execution, if any character is in motion, it can check during *each* frame whether or not it risks colliding with another character; it also may need to update joint angles in each frame.

Each character, gets an opportunity to set its own plan. We can keep track of the amount of time that has passed after each planning, and if after Ti we are out of our allocated time, we exit, and then in the next frame we can start the planning at Character i.

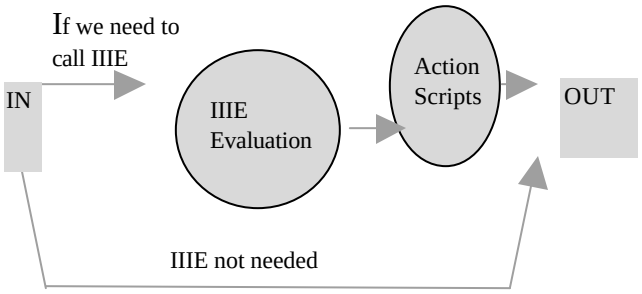

Figure 3. Set Plan Character i

If nothing important has changed (no "hot variables" set), Set Plan Character i exits. Otherwise, we evaluate the inference engine in order to see if we need a new plan. The Inference Engine returns an action token, which points to an action script. The action script is used to fill out the Character plan data structure for the given Character (to be described later).

Next, Execute Character i performs a round-robin loop. Each Character is given some amount of time to perform its tasks. In executing each Character's plans, we look at all active plans, and execute the next step of each of them (a step consists of a number of atomic actions that must be initiated simultaneously, such as a motion, a sound, some low-level planning). In the current implementations, some characters in the game have a limit of one script, but a WORLD entity can have many concurrent scripts.

This system emphasizes an object-oriented design to the characters. This gives us independent decision making for the characters and easy modification to one character's behavior (without affecting others'), but it still allows us to coordinate characters through the IIIE.

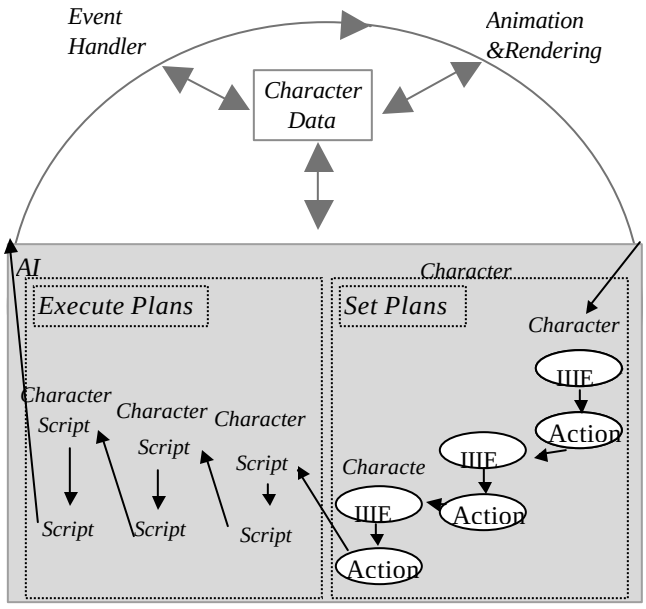

Figure 4. The Complete Flow of Game Control

Ian's Improved Inference Engine

The IIIE (*triple-I E*) evaluates potentially complex rules very quickly. The rules (at present) are straightforward conditionals, and there is no implicit chaining of rules in a single inference engine evaluation, although chaining can be accomplished in multiple steps. See [Tanimoto87] & [Charniak86] for general descriptions of Inference engines. Principal features of the IIIE include:

- *Compiled Rules*. The rules compile into and execute as dynamically loadable C++ code, improving the speed of evaluation and allowing use of commercial debugging tools.
- *When-Needed Rule Evaluation*. We use variables call "hot variables". During pre-compilation, each hot variable is linked to all of the rules that depend on it. During the execution of the game, if any hot variable's value is modified, it is marked as dirty. When the IIIE evaluation happens for each character, only the rules that reference dirty hot variables are evaluated.
- *Arbitrarily Complex Logic*. The Left-hand sides of the rules can include arbitrarily complex arithmetic & logical expressions
- *Simple Grammar.* The raw rules and scripts provided by the designers are easy to read and write. This example shows a rule that compares a "hot variable" (SELECTED_OBJECT) to the ID of a red button, and calls an action script that animates some special effects:

```
RULE red_button_pushed

if (SELECTED_OBJECT is red_button_obj)

then
 explode

END_RULE
```

- *Object Orientedness*. Each character maintains its own separate knowledge base (rules & scripts), but can also share mission specific data

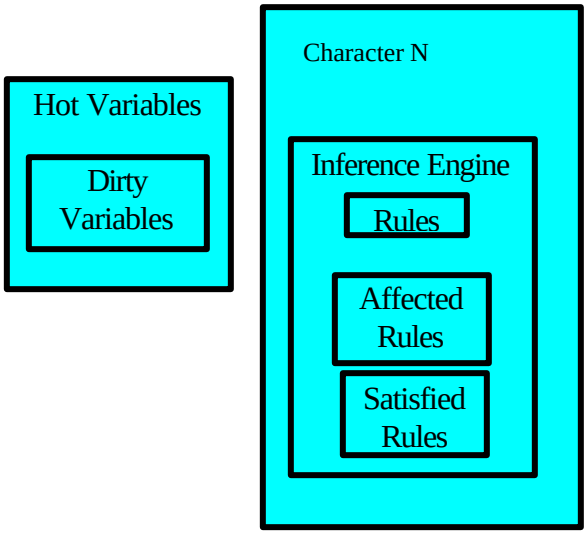

Figure 5. The IIIE Organization

All of the logic in the IIIE & CCA is generated from rule files and action script files. These are written in a designer-friendly grammar, and are preprocessed into C++ code that is compiled into loadable libraries (Dynamic Link Libraries, or DLLs). In these files, the user can define characters, action scripts, variables, and rules (though characters can also be generated in the game core). The rules are specific to each character, while the action scripts can be accessed by all of the characters.

The basic measurement of time is the cycle (also known as the frame or loop). If, in a given cycle, none of the hot variables have been changed (a dirty flag in each variable signals a change), the CCA does nothing. Only when a variable has been modified does the CCA put the relevent rules on the Affected Rules List (ARL). At each cycle, the CCA uses the ARL to modify the SRL (Satisfied Rules List) for each character. The SRL consists of all of the rules whose left-hand sides are currently true. Initially, the SRL is empty, but any rule from the ARL whose left side evaluates to true is added to the SRL (unless it is already there). Furthermore, any rule from the ARL whose left-hand side is now false can get removed from the SRL.

This guarantees that at each cycle we have a list of satisfied rules to choose from which is complete and which was generated efficiently. We choose (either at random or through a priority ranking) a rule from the SRL as the current rule to fire. Firing the rule means calling the right hand side of the rule (an action script which has been turned into C++ code). This entire process repeats for each character under the control of the CCA.

Examples

In order to give a more complete understanding of this system, we are including sample rules and scripts for one small part of a level in Planetfall 2: The Other Side of Floyd (as well as showing a sample debugging trace of the rules). In this setting, the player is locked into a cabin aboard a spaceship, and Floyd and Oliver need to help extricate him. The rules show how the IIIE lets Floyd be used for both storytelling and as part of the player's arsenal of tools. Note that for space considerations, we can only show some of Floyd's rules and scripts. Furthermore, we cannot explain in detail what each atomic action means, though most involve moving the characters or playing sounds. Two of Floyd's rules shown here are responses to commands given by the player, and two are rules Floyd follows when not being bothered.

Sample Rules

Here are a few sample rules for Floyd. They handle his autonomous actions, as well as being selected by the player to report in or open a door.

```
// Rules for Floyd for the IIIE Inference Engine & CCA
// for use with the demo scenario "Paxton's Room".
RULES FOR FLOYD


//
// IDLE_MOTIONS
//
RULE idle_motions
if ((FLOYD.STATE is IDLE) and (next_floyd_idle is WALK))
then
   floyd_pause
or
   floyd_wander
END_RULE

//
// IDLE_GAGS
//
RULE idle_gags
if ((FLOYD.STATE is IDLE) and (next_floyd_idle is TALK))
then
   floyd_generic_sound(how_long_sound)
or
   floyd_generic_sound(floyd_ID_sound)
END_RULE
```

```
//
// REPORT_IN
//
RULE report_in
if (SELECTED_OBJECT is floyd_obj)
then
   floyd_here
END_RULE

//
// OPEN_DOOR
//
RULE open_door
if ((SELECTED_OBJECT is RIGHT_DOOR) and
(gCharacterIcon is FLOYD_ICON))
then
   floyd_open_door
END_RULE
```

Sample Scripts

The corresponding Action Scripts follow (with many game-specific atomic actions). "Actions" are what are invoked by the rules firing. Within each action we can define scripts for any number of characters, although here we only show scripts for Floyd. Also note that within each script, several steps can be defined, and all atomic actions in one step have to be completed before the next step of the script will be executed. Most of the atomic actions – all lines between "SCRIPT FLOYD" and "STEP" or between "STEP" and "END_SCRIPT" -- get translated directly into C++ code (hence the syntax and commenting style) that's imbedded in the script's.

```
//
// floyd_open_door
//
// Move Floyd to the door.  Then open it!
ACTION floyd_open_door
   SCRIPT FLOYD
      FLOYD.STATE = BUSY;
      // By initiating an action, the request is satified
      FLOYD.PLAYER_REQUEST = NO_REQUEST;
      SELECTED_OBJECT = NONE_SELECTED;
      SetCursor(13);
      SetAnim(FLOYD,1,4,0);
      TurnToObj(FLOYD, A_HOTSPOT, 1);
      MoveToHotSpot(FLOYD, 1,4, right_door_spot,7.2,1);
   STEP
      SetCursor(0);
      SetAnim(FLOYD,1,5,0);
      SoundPlay(FLOYD, floyd_grunt,CYCLE_ONCE,
                  WAIT_FOR_SOUND_TO_FINISH);
      FLOYD_TUGGING_ON_DOOR = TRUE;
   END_SCRIPT
END_ACTION

//
// floyd_here
//
// Floyd's been clicked on, so say hi to player
ACTION floyd_here
   SCRIPT FLOYD
      SELECTED_OBJECT = NONE_SELECTED;
      FLOYD.STATE = BUSY;
      SetAnim(FLOYD,1,10,1);
      TurnToObj(FLOYD, THE_CAMERA, 0);
      SoundPlay(FLOYD,floyd_here_sound,CYCLE_ONCE,
                  WAIT_FOR_SOUND_TO_FINISH);
   STEP
      FLOYD.STATE = IDLE;
   END_SCRIPT
END_ACTION

//
// floyd: floyd_pause
//
// Floyd is bored.  Just shrug shoulders
ACTION floyd_pause
   SCRIPT FLOYD
      FLOYD.STATE = BUSY;
      SetAnim(FLOYD,1,3,1);
   STEP
      FLOYD.STATE = IDLE;
   END_SCRIPT
END_ACTION

//
// FLOYD: floyd_wander
//
// Choose some random spot and walk over.
// Let's set it up so we make a snide comment when we get there.
ACTION floyd_wander
   SCRIPT FLOYD
      int target;
      target = RAND_INT(6,9);
      FLOYD.STATE = BUSY;
      SetAnim(FLOYD,1,4,0);
      TurnToObj(FLOYD, A_HOTSPOT, target);
      DEBUG_MSG(debug_file, "\nTARGET: %d\n", target);
      MoveToHotSpot(FLOYD, 1,4,target,7.2,1);
   STEP
      next_floyd_idle = TALK;
      FLOYD.STATE = IDLE;
   END_SCRIPT
END_ACTION


//
// FLOYD: floyd_generic_sound
//
// Just make a rambling Floyd comment.
ACTION floyd_generic_sound(int which_sound)
   SCRIPT FLOYD
```

```
    FLOYD.STATE = BUSY;
    SetAnim(FLOYD,1,6,1);
    TurnToObj(FLOYD, THE_CAMERA, 0);
    SoundPlay(FLOYD, which_sound,CYCLE_ONCE,
                WAIT_FOR_SOUND_TO_FINISH);
  STEP
    next_floyd_idle = WALK;
    FLOYD.STATE = IDLE;
  END_SCRIPT
END_ACTION
```

Sample debugging trace:

A partial debugging trace of a 3 minute run through this room (edited for brevity and just to show Floyd's rules) is shown here. At the first IIIE run for Floyd, he's told to open the door for the room. In the second run, he's finished his actions and is restored to a waiting state. In the third, he gets bored and wanders off.

```
DIRTY VAR: SELECTED_OBJECT: 52

Running IIIE for FLOYD

# of Affected rules: 2
     Affected Rule: open_door
     Affected Rule: report_in
# of Satisfied rules: 1
     Satisfied Rule: open_door
RULE CHOSEN: open_door

  ACTION: floyd_open_door



DIRTY VAR: SELECTED_OBJECT: -1
DIRTY VAR: FLOYD.PLAYER_REQUEST: 0
DIRTY VAR: FLOYD.STATE: 1

Running IIIE for FLOYD

# of Affected rules: 4
     Affected Rule: open_door
     Affected Rule: report_in
     Affected Rule: idle_gags
     Affected Rule: idle_motions
# of Satisfied rules: 0


DIRTY VAR: FLOYD.STATE: 0

Running IIIE for FLOYD

# of Affected rules: 2
     Affected Rule: idle_gags
     Affected Rule: idle_motions
# of Satisfied rules: 1
     Satisfied Rule: idle_motions
RULE CHOSEN: idle_motions

  ACTION: floyd_wander
```

Conclusions

In designing our system, we had a few goals for it: it had to be fast, flexible, easy to debug, and relatively easy to write rules for. We seem to have achieved much of this:

The IIIE & CCA allow us to attach just the needed set of rules to each character and to evaluate exactly and only the ones we need. Because the rules are compiled, even on a 133 mHz machine we can evaluate over 25000 rules per second (with test sets that required all rules to be evaluated at all times). However, because of the "hot variables" and when-needed rule evaluation, we rarely needed to evaluate more than a few dozen per second. For *Planetfall 2: The Other Side of Floyd*, we anticipated needing 20-30 rules loaded concurrently for non-central characters, and 30-100 rules for central characters. In our current 3D RTS game, we have up to 100 rules per team for high-level decision-making, and anticipate 5-20 rules per ship (with up to 40 ships total) for tactical decisions. The core system is virtually unchanged for our RTS game, which demonstrates the flexibility of the system. Additionally, the relatively straightforward grammar (strangest only when it's most like C and C++) has allowed non-programmers to learn the system rapidly. Finally, our compiled rules allow debugging using commercial tools such as symbolic debuggers.

It is worth noting that there are still many difficulties in rule design for any expert system. The designers must be carefully trained to watch out for conflicting rules, oscillating rule firings, and efficiency issues. With the combination of using the source level debuggers for the pre-processed rules and scripts that have been turned into C++ code, and watching rule evaluation traces, we have managed to have designers with only rudimentary programming training create complex missions and behaviors. We anticipate making several additions in the near future to help with rule design, including the ability to define mutually exclusive subsets of rules (such as "Floyd Speaking Rules" or "Space Destroyer Targeting Rules") and rule priorities which will be used to decide between rules in mutually exclusive sets.

For now, the CCA and IIIE system has proven itself to be fast, easy to use, and flexible enough for both a 3D action game and a Real-Time strategy game. We hope to apply the system to many more upcoming games.

Acknowledgements

The author would like to thank Martin Martin, Brian Hawkins, Linus Chen, and Ken Miller for their contributions to the CCA and IIIE, Ryan Kirk, Eric Gewirtz, and Mike Ward for their rule scripting, and Scott Lahman, VP of Production for supporting advanced AI work at Activision and for encouraging publication.

References

[Charniak86] E. Charniak and D. McDermott, *An Introduction to Artificial Intelligence*, Addison-Wesley, 1986.

[Tanenbaum87] A. Tanenbaum, *Operating systems: Design and Implementation*, Prentice Hall Press, 1987.

[Tanimoto87] S. Tanimoto, *The Elements of Artificial Intelligence*, Computer Science Press, Rockville, Maryland, 1987.