# Rule Databases for Contextual Dialog and Game Logic
## *or..*
# How To Make Writers Even More Awesome

**Elan Ruskin**
Valve Corporation

Good afternoon everyone! Thanks for sticking with us all the way to this, the last talk in the last hour on the last day of GDC.

I'm Elan Ruskin and today I'm going to talk at you about dialog in games, and how to build a system that lets your writers make it as they do best.

Hard to keep up taking notes?

Taking pictures of every slide?

Relax, slides will be at:

　　　valvesoftware.com/publications.html

And my blog:

　　　assemblyrequired.crashworks.org

# What is this talk about?

- A system for tracking lots of facts and then picking a rule from a database based on matching criteria
- Originally used for character speech in Half-Life 2, Team Fortress 2, Left4Dead, Portal 2, Dota2, …
- Can also drive animation, sound, AI, gameplay logic, and many other things

Today we're going to talk about a system we have for tracking a whole bunch of state about the world in a uniformly manageable way, and then using that state to select just the right line from a big database of character speech. It's the system we used in The Orange Box, Left4Dead, Portal, Dota…. All our games. You can use the same mechanism to drive things other than character dialog, although that didn't occur to us until after we had built it.


..
Who is this talk for?
        Programmers and writers.
                Programmers will see how to build a system for creating AIs that dynamically generate dialog based on world state, and possible extension into other fields via script.
                Writers will see the possibilities of such a system, and suggestions for how to create an interface that allows them to generate content easily.
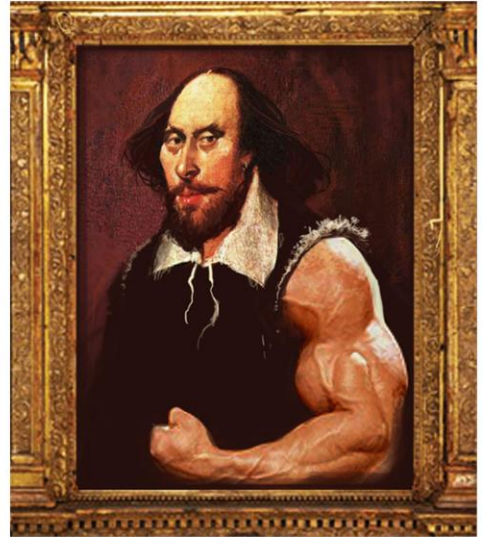What's this about?
        Dynamic game dialog, the code behind the engine that drives ours, and thoughts on how to design games around such dialog and dialog around games.
        And then how you can extend the system to drive other things

# What is this talk about?

- Empowering Writers!
- A philosophy of character dialog and behavior
- Tracking lots and lots of context to drive gameplay



*Shakespearicles, the strongest writer who ever lived*

But more importantly today's talk is about empowering writers. You can't have dialog without writers, and making writers' lives easy is the best way to get good dialog. So this is mostly about how to generally think about world state and character behavior in a way that enables writers to create and iterate independently.

# Response Rules: context tracking that is

1. Simple
2. General
3. Efficient
4. Friendly

**SIMPLE = FRIENDLY**

We've tried to make a system that is as simple as possible, general enough for many features in all our games, efficient enough to be used at runtime, and user-friendly for the writers who populate it. This is actually three things, not four, because I firmly believe that simple and friendly are the same thing. A system that is simple to use gets used. A system that is simple to write gets maintained.

# Why talk about dialog?



Why stand here and talk for an hour about talking? Well, there have been a bunch of games recently that I think did interesting and successful things with dialog that…
… was an artistic element of the game
… remembered and responded to the player's actions
… created character and environment
… or was just plain fun.

I don't know how they did their stuff, but I know how we did it, and I think our system could enable people to make more games like these. I liked all these games, so I want people to make more like them, so I can play them! My ulterior motive.

# The Problem of Contextual Dialog

- *ie* dialog that responds to player actions and world state
- In the beginning this was simple

```
courtyard                                           Time: 7:04 pm
The front gate closes and locks behind you.
(You are now in the courtyard.)
As flood lights blaze on, you look around. It looks even lovelier than it
sounds in the tourist brochure.
The dark stone turrets rise toward the misty sky.
Your new little green sports car is parked here.

Someone comes running out of the wing to greet you. She's a beautiful
red-haired young woman of average height. You recognize her as your friend,
Tamara Lynd.
"Moby!" she cries with outflung arms. "You sweet thing, to answer my letter in
person this way! And all the people I wrote about are here tonight for Lionel's
memorial birthday dinner!"
After a warm hug, she asks anxiously, "You did read my letter, and not just
give it a hasty glance?"

>YES
"Then you know about my engagement, and the White Lady, and the fact that...
that someone is trying to kill me!"

[You won't see "What next?" any more.]

>YES
```

*Moonmist* (Infocom, 1986)

By "contextual" or "dynamic"' dialog I just mean any system where characters speak, and the speech seems to respond to players' actions, the state of the world around them, and previous events. In the beginning, when games were nothing *but* dialog, this was pretty straightforward.

# The Problem of Contextual Dialog

- But it quickly got complicated
- Some games used trees to create choice, and stored explicit flags
- But that gets unwieldy pretty fast



*Fallout*

Things got complicated pretty fast after that. Early RPGs introduced the notion of conversation trees, so players could have back-and-forth with characters, but those got big. Also, you might want a character to remember the fact that you blew up their home village three missions ago, and pick a different tree based on that, so that entailed keeping a bunch of global state and control flow which gets unwieldy very quickly.

# The Problem of Contextual Dialog

- Some games have characters "bark" based on what the player does
- Example: stealth games
- Used to convey AI state



*"What's in that shadow over there? I'd better check it out."*

Some games don't have conversation trees, but they have AI charactgers who "bark", or speak spontaneously, to announce their state or forward the story. For stealth games this is a critical game mechanic, because a big part of the game is the player being aware of what the AI knows, what it's planning on doing, and generally keeping track of the AI's state of mind. Having an AI that clearly communicates its intentions is a critical part of the game.

# Conveying AI state



A great example of this is Batman: Arkham City. (I *love* that game. So much good stuff in it!)
This game has a bunch of different characters who convey state in stealth sequences. One of them is so clear and straightforward that it's like it was tailor made for demonstrating!

---

> (video) Example: the TYGER guards in Batman Arkham City, who actually say things like "Target has been sighted!" "Converging on last known location!" "Enemy object found! Initiating search!" "Scanning in dark areas!" and so on.

# Conveying AI state



A great example of this is Batman: Arkham City. (I *love* that game. So much good stuff in it!)
This game has a bunch of different characters who convey state in stealth sequences. One of them is so clear and straightforward that it's like it was tailor made for demonstrating!

> (video) Example: the TYGER guards in Batman Arkham City, who actually say things like "Target has been sighted!" "Converging on last known location!" "Enemy object found! Initiating search!" "Scanning in dark areas!" and so on.

# Used as an artistic or narrative device



"Ground forms up under his feet as if pointing the way… he don't stop to wonder why."

*Bastion*
(Supergiant, 2011)

Some games use contextual speech as an artistic device. Bastion has an omniscient narrator who seems to respond to everything the player does. That's not because the player needs the game to tell him what he's doing, of course, but because that's what the game is *about* -- storytelling, and creating a world from your actions.

# Some games need more than if-else



Sports commentators generate thousands of potential lines for highly dynamic circumstances.

*Madden 2011* (EA)

Sports games have elaborate commentators that somehow assemble tens of thousands of possible utterances from thousands of individual snippets about this hugely dynamic environment that can get into I don't know how many possible situations. Handling all of this by just a forest of if-else would be painful.

# Some games have *lots* of state



*Skyrim* (Bethesda)

MASS EFFECT 2

- Player history and world events factor into dialog choices, character outcomes.
- Varied, context-sensitive speech possibilities for "townsfolk" outside converation trees
  - suggest a living world
  - prevent characters from falling back on generic, repetitive lines.

And think about modern RPGs and all the state they track! Mass Effect and Skyrim have to remember a huge library of things that the player's done all over the world. It affects conversation possibilities and the character stories avaialble; but it's also useful to create the feeling of a living world from just the passerby barks. If ordinary townspeople remember that you are the elven mage that saved their village from the demon horde, and interact with you differently based on that, you feel much more like a part of the world than you would if they just said the same canned line over and over.

# There are many systems, this one is ours.

Like I said, I don't know how they implemented their stuff, but here's the way we do it. I think it would work pretty well to make games like those, and maybe it'll work well for you.

# One simple, uniform way to…

- Track lots of state in a dynamic game
- Find the right line for every circumstance
  - Special lines for specific cases
  - General lines for all the rest
- Allow writers to create and iterate a powerful rules-driven system
  - Programming a book of "rules" with "criteria"
  - Not dense if/else trees and function calls!

Our system was designed to be simple – SIMPLE – and uniform. We wanted one straightforward way to track all the state of a game that could possibly be used to select dialog, and a convenient way for writers to specify which pieces of state select which bits of voice. We want writers driving as much of the dialog-creation process as possible, because… that's what they're there for.

So, a quick bit of history about the system's early origins. Team Fortress 2 is a multiplayer game where players choose from one of nine character classes. Each of the characters has its own voice. So, we have a mechanism for allowing players to communicate in the voice of their character.

# Team Fortress 2: player-triggered lines



If, for example, I'm playing a soldier character and I'm injured and I need medical attention, I can hit a button on my keyboard to say that in the voice of my character.

("MEDIC!!!")

Allowing players to communicate simple orders to each other in their characters' voice encourages roleplaying and immersion, but it also helps international play. If I'm playing in the United States, I'll hear my soldier speak English, but if you're playing with me from Spain, then you'll hear his localized voice in Spanish. So this is a way for players to communicate with each other across language boundaries.

# Team Fortress 2: player-triggered lines



"Need a dispenser here!"

# TF2: automatic state announcement



Other times you want characters to automatically announce important state without the intervention of the player. For example, if I'm playing near a medic, and the medic is ready to deploy his uber-heal-ray, I need to know about that whether the player controlling the medic remembers to tell me or not. Having characters announce such state on their own is an important prompt to players about actions they may need to take.

# TF2: automatic state announcement



"We must stop tiny cart!"

# TF2: automatic state announcement



"Sentry going up!"

# TF2: player triggered with context



- Player clicks generic "vocalize" button

We also had a mechanism for contextual player-initiated dialog – basically you can put your cursor over an object in the world, hit the "vocalize" button, and have the game try to figure out what you probably mean and have the character say it in their voice. This is sometimes more convenient than speaking at length, and it allows a bit of role-playing.

# TF2: Contextual vocalize button



It's also an opportunity for us to play up allegiances and rivalries between characters in the game world – scout vs heavy, sniper vs spy, for example; the character voices gibe at each other and create a bit of storytelling automatically while the players just play their game.

"the enginer is a spy!"

# TF2: Contextual vocalize button



"the scout is a spy!"

# TF2: Contextual vocalize button



"The spy is... a double agent!"

# An internal design experiment

- World premiere of *almost never before seen content*

**TWO BOTS,
ONE WRENCH**

(don't ask)

Voices by Richard Lord, Iestyn Bleasdale-Shepard

After The Orange Box,  Valve did something kind of… Valve.
We'd just finished this big (cluster of) games, and we weren't quite ready to go into production on the next one. So we decided to take a little time to come up with new ideas. We divided the old team into lots of little groups, like 3 or 5 people apiece, with each group trying to come up with some new design idea, some feature, some experiment that we could roll into a future game. It was a chance to try out risky ideas.
One team chose to explore companion characters – what could we do to make friendly characters who seemed more aware of their environment and the player? Who had memories and felt like they grew along with the player's experience? We took a bunch of Half Life art assets, built a couple of character models, had an artist and a programmer supply voices, and came up with …. TWO BOTS ONE WRENCH.

I was going to say "here, an internal project shown for the FIRST TIME EVER at GDC," but I've just been told that Geoff Keighley has shown these assets once before. So here, for the… second time ever… the making of an internal design experiment.

# Environment-aware speech

- Objects are tagged by name (eg "soda_can", "radiator")
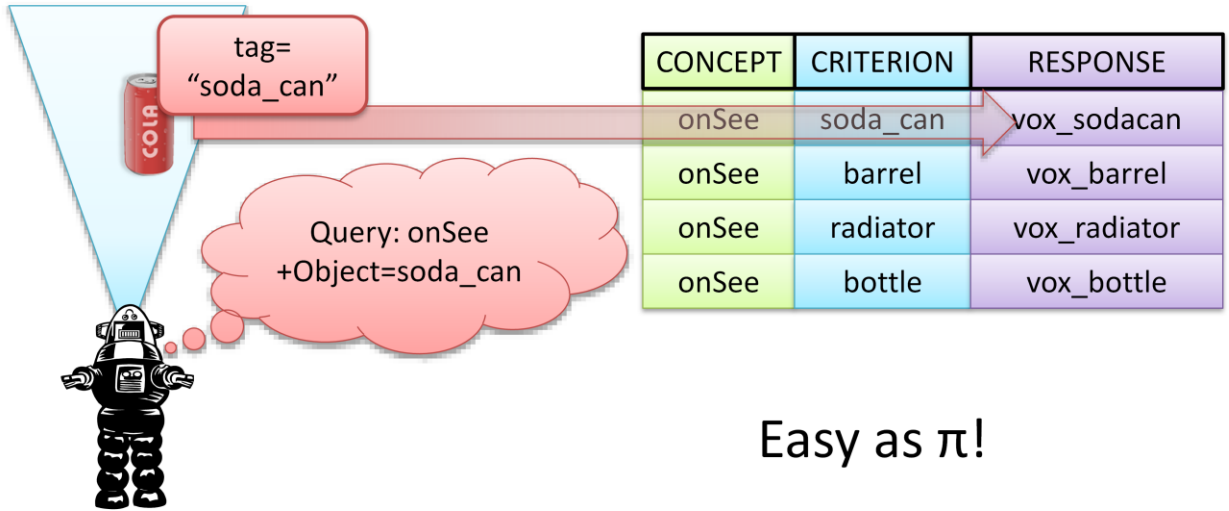
Tag = "soda_can"

Tag = "barrel"

Tag = "bird"

First we asked, "what's the easiest, simplest way we can prototype making characters aware of their environment?"

Well, we had that code from Team Fortress that we used to find contextual dialog based on what the player's cursor was over.

So we exploited the same tech. We marked each object in the world with a unique string class name.

# Environment-aware speech



tag= "soda_can"

Query: onSee +Object=soda_can

| CONCEPT | CRITERION | RESPONSE |
|---------|-----------|----------|
| onSee | soda_can | vox_sodacan |
| onSee | barrel | vox_barrel |
| onSee | radiator | vox_radiator |
| onSee | bottle | vox_bottle |

Easy as π!

Then each robot simply polls its field of vision every few seconds. If it finds an object there, it tries to find a line in its database corresponding to the object's tag. If there's a match, it plays.

Super simple.


Objects are tagged by name (eg "soda_can", "radiator")
Database stores possible lines indexed by object name
Characters poll their field of vision for objects in world. If a line matches, say it.
Easy as π!

Video available via
http://assemblyrequired.crashworks.org/gdc2012-dynamic-dialog/

# Environment-aware speech

*For each character:*
- *once per ten seconds, find objects in vision cone*
- *select one object and trigger a 'SeeObject' speech concept*
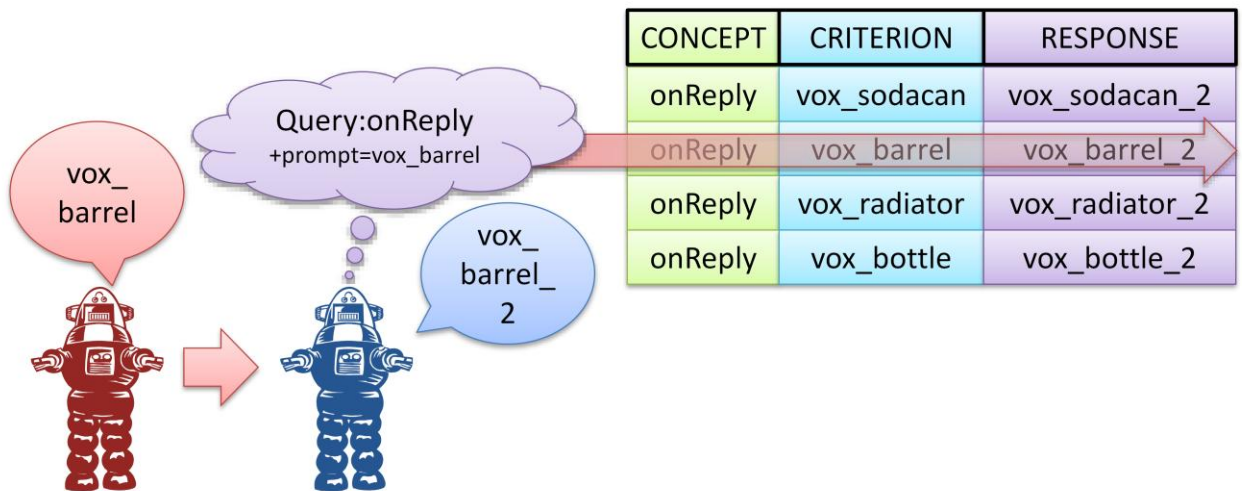
```
rule BluBot_ObjectSee_Cart
{
        criteria                ConceptSeeObject NotInCombat ObjectName=LaundryCart
        response                BluBot_See_LaundryCart // there's a laundry nearby!
}

rule BluBot_ObjectSee_Shoe
{
        criteria                ConceptSeeObject NotInCombat ObjectName=Shoe
        response                BluBot_See_Shoe // that's a shoe!
}

rule BluBot_ObjectSee_Crate
{
        criteria                ConceptSeeObject NotInCombat ObjectName=Crate SeenForklifts=0
        response                BluBot_See_Crate // how did this crate get here without a forklift?
}
```

This is all the code it took to pull it off. Script, really – we call them response rules. To create a new line about an object, a writer just needed to add a new stanza to this in Notepad, with an additional criterion for the object's tag name. If there wasn't a line for an object, nothing plays. Since we poll every object, adding a new bit of dialog just means adding a new rule. Four lines of text – one of them is the name of the vox file to play.

# Starting a conversation



| CONCEPT | CRITERION | RESPONSE |
|---------|-----------|----------|
| onReply | vox_sodacan | vox_sodacan_2 |
| onReply | vox_barrel | vox_barrel_2 |
| onReply | vox_radiator | vox_radiator_2 |
| onReply | vox_bottle | vox_bottle_2 |

Walking around reading signs is all well and good, but we wanted characters to talk to each other. And there's lots of ways that you can handle conversation in games – create scripted sequences, entities that lock down the two characters, some kind of purposebuilt statekeeping for conversation. But that's a lot of work, and I'm always in a hurry. So once again, we asked, what's the simplest possible way we can do this? And we figured that the same way we had the robot poll its vision every few seconds, and send itself a "onSee" event with the name of the object if something were there; we could have each line of dialog by the red robot dispatch an "onReply" event to the blue robot when it was finished. The same way we paramterized onSee with the object name we paramterize onReply with a unique tag for the bit said by the first robot. If the second robot has a reply, it plays; possibly it triggers a reply on the first robot, and back and forth. Again, brain-dead simple.

//
Every line said by a bot *also* gets a name tag, *eg* "redbot_danger_flammable"
When red bot finishes speaking, automatically triggers a lookup on the blue bot
If blue bot has a rule in its database matching the redbot's followup tag, then it plays.

# Starting a conversation

Video available via
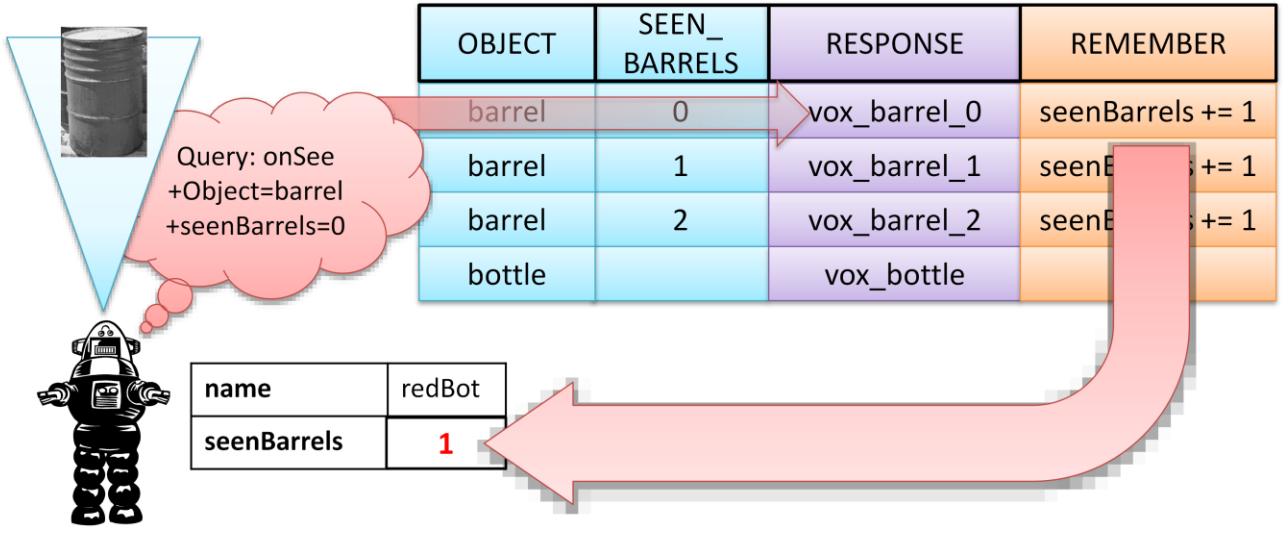http://assemblyrequired.crashworks.org/gdc2012-dynamic-dialog/

# Memory and Context

- Rules can also *write context* back to the character or the world

- This context is appended to the next query performed by the character.

- Creates *memory*, the ability to pick subsequent lines based on what happened before.

Well, if you've got conversation, then you *need* running gags. We wanted companion characters that had memory; that reacted differntly based on what the player had done near and to them before. Again, what was the simplest, easiest way we could do this?

We figured we could add just one more little bit of technology. The same way all the previous rules were parameterized by *eg* the object seen, along with other criteria; we figured we could create a table of "memory" in the character's head and then send that along with every voice query it made.

# Memory and Context



| OBJECT | SEEN_BARRELS | RESPONSE | REMEMBER |
|--------|--------------|----------|----------|
| barrel | 0 | vox_barrel_0 | seenBarrels += 1 |
| barrel | 1 | vox_barrel_1 | seenB  s += 1 |
| barrel | 2 | vox_barrel_2 | seenB  s += 1 |
| bottle |  | vox_bottle |  |

Query: onSee
+Object=barrel
+seenBarrels=0

| name | redBot |
|------|--------|
| **seenBarrels** | **1** |

So here you can imagine that we have a bunch of rules where the matching criteria are not just "what am I looking at?" but also an arbitrary "seen_barrels" variable. The first time a robot sees a barrel, the first rule matches. It plays the first line, but it also sets a bit of state back in the robot's head.

Photo of barrel via Wikimedia Commons.

# Memory and Context

| OBJECT | SEEN_BARRELS | RESPONSE | REMEMBER |
|--------|--------------|----------|----------|
| barrel | 0 | vox_barrel_0 | |
| barrel | 1 | vox_barrel_1 | seenBarrels += 1 |
| barrel | 2 | vox_barrel_2 | seenBarrels += 1 |
| bottle | | vox_bottle | |

Query: onSee
+Object=barrel
+seenBarrels=1

| name | redBot |
|------|--------|
| seenBarrels | **1** |
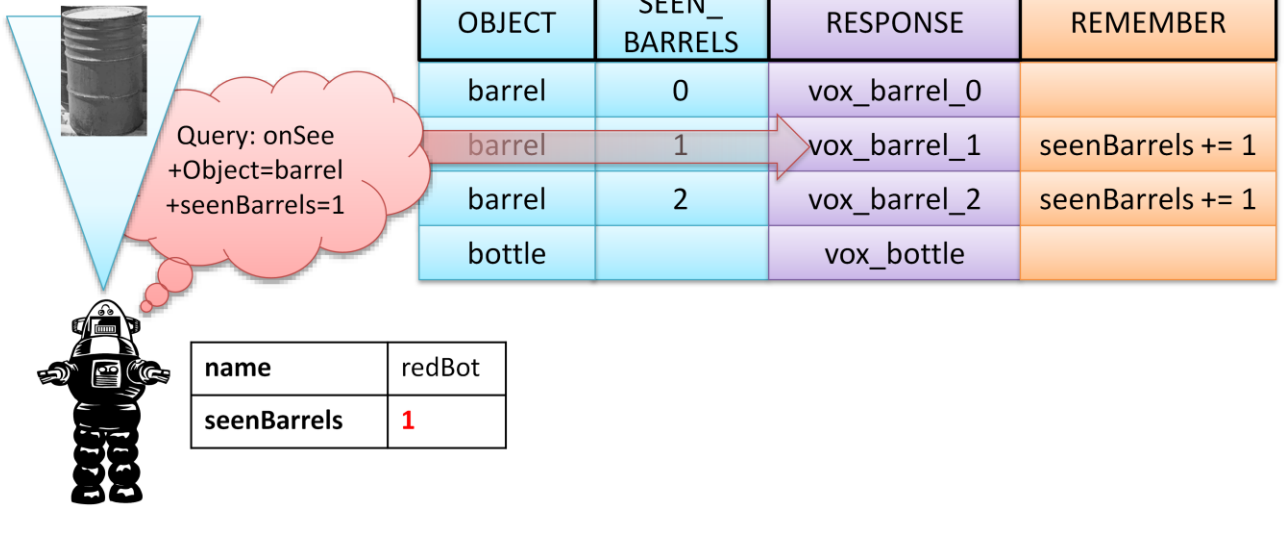
So the next time the robot sees a barrel, it matches the second rule in the database –
the one with a different seen_barrels criterion. The criterion name is arbitrary; it's
just what the writer chose to name the variable in the robot's head.

# Memory and Context

```
rule BluBot_ObjectSee_Barrel_A
{
          criteria              ConceptSeeObject NotInCombat ObjectName=Barrel SeenBarrels=0
          response              BluBot_BarrelA        // hey look, a barrel!
          remember              SeenBarrels:=1
}

rule BluBot_ObjectSee_Barrel_B
{
          criteria              ConceptSeeObject NotInCombat ObjectName=Barrel SeenBarrels=1
          response              BluBot_BarrelB        // a second barrel!
          remember              SeenBarrels:=2
          trigger               RedBot  SeenBarrel
}

rule RedBot_Friend_Saw_Barrel
{
          criteria              ConceptSeenBarrel From=BluBot NotInCombat
          response              RedBot_Barrel_Reply   // what's with all these barrels?
}
```

"Then" rules can also *write context* back to the character or the world
This context is appended to the next query performed by the character.
Creates *memory*, the ability to pick subsequent lines based on what happened
before.

# Memory and Context

Video available via
http://assemblyrequired.crashworks.org/gdc2012-dynamic-dialog/

# Why Left4Dead (1 & 2) needed more



VS

100×

Left4Dead is a game about four people – you and your friends – fighting against a zombie horde. It is designed for replay, so you'll go through each campaign many many times. That means we need wide variety; otherwise if you hear the exact same canned lines at the exact same points over and over, it becomes dull very quickly.

# Why Left4Dead (1 & 2) needed more



We have a complex AI "director" to create that variety, by dynamically responding to player actions, throwing different enemies at them each time, and generally trying to keep it fresh.

You can see why a basic system of brush triggers playing canned lines won't work here. For one thing, events don't always happen in the same place in the same order; you can't have Nick, the Gambler, play a line about the hunter-zombie every time he walks into the warehouse, because it may not be there on a given playthrough. Also, with each level played so many times, that degree of repetition would be really painful; you need much more variety in the placement and nature of the canned speech.

# Why Left4Dead (1 & 2) needed more



And not every survivor makes it to the end! At any point in the map, you may have only some subset of the survivors with you; the others may be dead or straggling. So the system needs to cope with having different sets of characters available at each point in the map.

# Variety



Here's an example of what I mean by variety. This is the exact same location of the exact same mission, but on two successive playthroughs.

Video available via
http://assemblyrequired.crashworks.org/gdc2012-dynamic-dialog/

# Left4Dead2 automatic barks



We also needed a way for the AI characters to shout out important facts in the world. If I'm playing with my friends, and one of them sees an ammo cache in a dark corner, then my friend can just tell me. That's an important thing to know; the ammo is in different places every time. But you don't always have four humans in a game; we have bots that fill in for missing players, and a single player mode. I still need the AIs to convey that kind of information as if they were humans. So we needed context-triggered speech that the bots could play to call out things like weapon caches, as if they were humans, and without level designer markup. Once we did it for the bots, we realized it was pretty cool for the human-controlled characters to do it also; it's a bit of additoinal roleplaying, and more convenient than having to call out yourself.

Video available via
http://assemblyrequired.crashworks.org/gdc2012-dynamic-dialog/

# Left4Dead2 environment-triggered dialog



Left4Dead is a game that tells its story through the environment. We don't have much in the way of cutscenes; the story of the zombie apocalypse is told through the things that you see around you as you move through the world. So, the best way we had for characters to tell their stories, to express who they are and show their development, was through having them remark on the environment also. That's a chance for running gags too.

Video available via
http://assemblyrequired.crashworks.org/gdc2012-dynamic-dialog/

# Scaling up and up!

- Left4Dead2 had ≈10,000 lines of dialog
- Four main characters interacting with
  - Each other
  - Environments
  - 12 special zombie subtypes
  - Map-specific characters / situations
  - Special game modes (survival, versus, etc)

The Left4Dead series had more dialog than any of our games to date – over ten thousand lines. That's moderate by RPG standards but it's more than we'd ever had in an FPS. So we also wanted a system that could manage script and speech at high scale.

# Give your writers tools, not homework

Assignment 2b: fill in the appropriate blanks.

1. "Help, a _____ is attacking _____ with a _____."

2. "I see a _____ in the _____."

3. "Get to the _____ !"

4. "I used to be a _____, but then I took an _____ in the _____."

5. "It is good to _____ your _____, to see them driven before _____, and to hear the _____ of the _____."

The important thing about dialog is not the code. It is the writers. Writers make dialog. That's what you pay them for!
In a lot of games, the AI programmer looks at all the events that happen to a character, and tries to guess, "well, for which events would this character want to say something?" The programmer puts in code hooks for each of the sites he can think of and then hands the writer a big spreadsheet and says, "okay, fill out a line for each of these events."

Well, the problem with this is you're basically reducing your writers to filling out a series of mad-libs. That kind of cramps their style. It also means that it's really the programmers doing the writing, and the intersection between the set of good programmers and the set of good writers is pretty small (unless you've got Vernor Vinge or Ted Chiang working for you).

Also, any time the writer finds a new circumstance in which she'd want to have a line, she has to go back to the programmer and ask for a new code hook to be put in. That's slow and it means less stuff gets written. So we really wanted a way for writers to decide which circumstances got lines and how finely those lines were specialized.

… and  I get to make the last arrow-to-the-knee joke of this conference.

# How It Works

*Everything should be made as simple as possible, but no simpler.*

— Albert Einstein  (probably)

Now let's go behind the curtain to see how it works. The basic idea is really simple. Like, head-slappingly "it's so obvious in retrospect" simple. That's what makes it user friendly.

Remember also that it grew by accretion. It wasn't designed; it kind of evolved as the writers made suggestions over the course of several games. If we'd set out to design something like this, we probably couldn't have come up with something so simple.

# The General Idea



| key$_1$ : value | | {k$_1$=v$_1$, k$_2$=v$_2$, k$_3$=v$_3$, ...} | "hello" |
| key$_2$ : value | | {k$_1$=v$_1$, k$_2$=v$_4$, k$_3$=v$_5$, ...} | "zombie!" |
| key$_3$ : value | | {k$_1$=v$_1$, k$_2$=v$_4$, k$_3$=v$_6$, ...} | <handwave> |
| key$_4$ : value | | {k$_1$=v$_8$, k$_2$=v$_2$, k$_3$=v$_3$, ...} | "kittens" |

*query*                              *rules*      *responses*

dialog

*character*
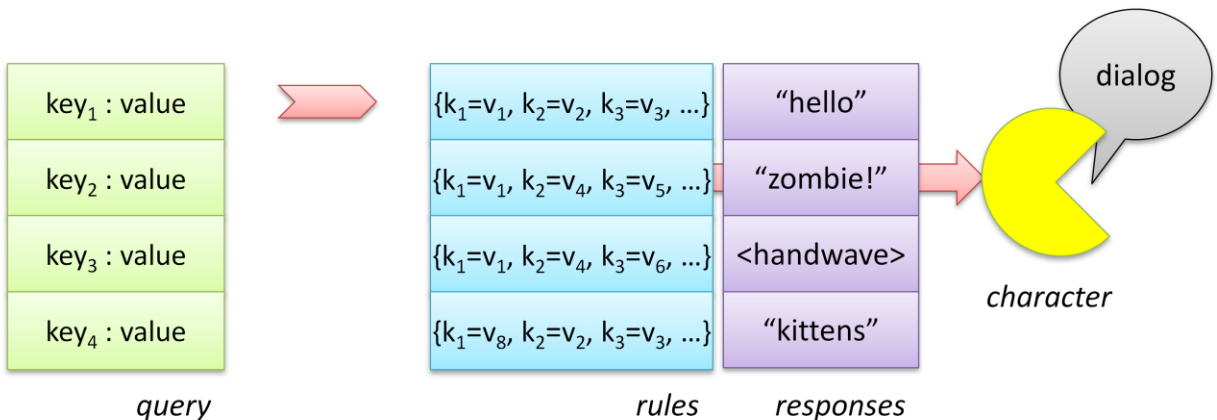
Here is the basic idea behind the system.

You have a database of rules, each with a list of criteria about the state of the world that must be met for the rule to be considered a match. A rule has an associated "response" which is simply the thing that happens when a rule matches, such as a voice file or an animation. When it is time to say a line, a query is constructed of many key-value pairs. The database searches for the rule with criteria best matching the data in the query; if one is found, the response is sent to the character, who utters it as dialog.

# Key concepts

- Context (aka "Fact")
- Query
- Criterion
- Rule
- Response

A "context" or "fact" is a piece of world state. Like "current map is swamp."
A query is a pile of those glued together. All the state of the world, used to lookup an action.
A criterion is a single function that returns true or false for a piece of world state. Like "zombies greater than 3."
A rule is a list of a criteria, all which must be true for the rule to "match."

# Thinking of the world as a pile of facts

```
if (
 ( this->name == "Protagonus" ) &&
 ( globals->GetCurMap()->name == "Cave Of Troglodus" ) &&
 ( globals->GetKilled( kENEMY_WOLF ) == 8 ) &&
 (  savedstate->Get("Town1")->Get("King")->m_isAlive ) &&
 ( !savedstate->Get("Town1")->Get("Cobbler")->m_isAlive ) &&
 ( player->GetInventory()->Get( "HammerOfSmiting" ) != NULL ) &&
 ( player->GetAllies()->GetNearest()->name == "Bob the Bludgeoner" ) &&
 ( world->FindEntitiesNear( player->GetLoc(),
          kTYPE_ENEMY ).count() == 3 )
 ( globals->Quests.Get(3)->m_isComplete )
 ( ((C_Orb*)
    (player->GetInventory()->Get( "MagicOrb" ))->GetCharges() == 12 )
)
```

Programmers are used to thinking of the world as a bunch of facts strewn hither and
yon. If I want to have an action that occurs when the player is in the cave and the
wolves have been killed and his ally is nearby and so on, then I can code it by building
a huge conditional intersection of a bunch of member variables and function calls.

There's a few things painful about this.

First, it's not discoverable what kind of data you have available to make a decision.
You sort of have to know a priori what information is present for you to test (typically
by having put it there yourself) and where to go looking for it. If you don't know that
there's a saved state object that tracks the lives of every person in every town, you
may not even think to make special cases that depend on that.

Also, it's nonobvious how you *get* information from all of these sources. There may
be a complex chain of members and functions between wherever you're writing your
code and the information you need for your logic.

Finally, it's just plain messy. Look at that. Confusing.

# Thinking of the world as a list of facts

```
Who                          = Protagonus
CurrentMap                   = "Cave Of Troglodus"
WolvesKilled                 = 8
Town1.King.Killed            = false
Town1.Cobbler.Killed         = true
Player.HasHammerOfSmiting    = true
Player.NearestAlly           = "Bob the Bludgeoner"
Player.EnemiesNearby         = 3
Quest3.complete              = true
MagicOrb.charges             = 12
```

It's much easier and more natural to think of the world as a flat pile of facts. If you always pull every piece of information together into a flat dictionary – use sub-namespaces if you like – it's always clear what you have available to select login upon; just look at the query. Also, it's easy to find an individual bit of state in this tree; it's just a key lookup. If you always pull all the state of the world into a flat representation every time you look for a rule, then it's very simple to add new and more specific rules: you don't need to remember which pieces of world state are available in queries under which system. You've always got the whole world at your fingertips.

Plus, it's just plain easier for non-programmers to think of state like this.

# Context (aka "Fact")

- <u>A piece of world state.</u>
- a key:value pair, such as:
- "who": "louis"
- "hitpoints":57
- "object_under_cursor" : "urinebarrel_2"
- Keys are strings
- values may be numbers, handles, or strings

An individual "context" or "fact" is just a pair – a keyname string, and a variant value
(any type). Like "hitpoints are 57".

# Context (aka "Fact")

- <u>A piece of world state.</u>
- a key:value pair, such as:
- "who": "louis"
- "hitpoints":57
- "object_under_cursor" : "urinebarrel_2"
- Keys are ~~strings~~ symbols
- values may be numbers, handles, or ~~strings~~ symbols

Don't *actually* use strings, of course. Use symbols or interned strings instead. I'm just using "strings" as a shorthand for "human readable unique identifier."

# Query

- A list of Facts used to select a rule.
- Facts collected into a long tuple of {k:v} pairs, like
  { "who":"louis", "hitpoints":57, "nearest_ally":"zoey", … }
- *ie* an associative array of keys to variant data.
- Typically hundreds of items long.
- Certain key names are special.
  - "concept" is used to select the general type of speech requested
  - "who" always indicates the speaking character

Thus a query "context" is essentially an associative array of keys to variant data.

Certain keys may have special meanings to the implementation – for example, we use "concept" to specify the general type of line being queried, like "saw enemy" or "on hit by bullets" or "player pushed context-sensitive button", and "who" to indicate the character performing the speech query *ie* which voice are we looking up. In our system every query must have at least those criteria, but mathematically they're like any other criterion.

# Building a Query



| concept | "OnHit" |
|---------|---------|
| attacker | "hunter" |
| damage | 12.4 |

```
CL4DPlayer::OnHit(
  CNPC * attacker,
  float fDamageHP )
{
  // (assume gameplay code here)
  // ...
  ResponseQuery query;
  query.add("concept", "OnHit");
  query.add("attacker",
            attacker->GetClassName() );
  query.add("damage", fDamageHP);
  Speak( query );
}
```

The contexts in a query are built up from many sources.

First is the function that actually starts the query. It creates the query object and populates it with the basic information of the event you want the character to talk about, such as the general type of line you're searching for and event-specific info.

# Building a Query

$f()$ ➕ .4DPlayer::

| concept | "OnHit" |
|---------|---------|
| attacker | "hunter" |
| damage | 12.4 |

| hitpoints | 78 |
|-----------|-----|
| ammo | 43 |
| nearest_Ally | "coach" |
| nearest_ally_dist | 733.0 |
| current_weapon | "Axe" |
| … | … |
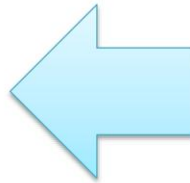
**Procedurally calculated at query time**

Then you call through to the base Speak() implementation, which starts to add in facts about the character who's speaking. This is when you procedurally pull in every fact that might be relevant to looking up speech, such as the character's health, weapon, nearby friends, any other local data or functions. The base Speak() member adds each of these to the table. You can chain through to ancestors' implementations as well, of course.

# Building a Query

*f*() ➕ [image] ➕ [image] 💭 memory

| concept | "OnHit" |
|---------|---------|
| attacker | "hunter" |
| damage | 12.4 |

| hitpoints | 78 |
|-----------|-----|
| ammo | 43 |
| nearest_Ally | "coach" |
| nearest_ally_dist | 733.0 |
| current_weapon | "Axe" |
| ... | ... |

| times_healed | 4 |
|--------------|-----|
| suit_complaints | 2 |
| zombies_killed | 204 |
| times_used_axe | 111 |
| times_used_shotgun | 93 |

⬅ **persistent store (as instance members / internal table/ etc )**

Then there's the persistent store inside the character, arbitrary data that can be written either by code or by writer-generated rules. This is where we store things like how many times a particular line has been said, events that happened previously, and so on. It's a table of arbitrary keynames and whatever the dialog rules have set. Add these to the associative array also.

# Building a Query

𝑓()  ➕  👤  ➕  👤  💭 memory  ➕  🌍

| concept | "OnHit" |
|---|---|
| attacker | "hunter" |
| damage | 12.4 |

| hitpoints | 78 |
|---|---|
| ammo | 43 |
| nearest_Ally | "coach" |
| nearest_ally_dist | 733.0 |
| current_weapon | "Axe" |
| … | … |

| times_healed | 4 |
|---|---|
| suit_complaints | 2 |
| zombies_killed | 204 |
| times_used_axe | 111 |
| times_used_shotgun | 93 |

| map | "swamp 2" |
|---|---|
| coach_alive | true |
| witches_killed | 7 |
| encountered_hillbilly_scene | false |
| total_zombies_killed | 1979 |
| … | … |

Then you merge in any procedural state about the world in general; current map, extant entity count, and so on. The world can have a persistent memory store as well, just like individual characters.

Take all of these sources of data, concatenate them all into one big associative array, and that's your query. It contains all the facts you'll use to select a line.

# Thinking of dialog as a pile of rules

*When seeing the player, say*
> "Look! It's Batman!"

*But if the player is in shadow, then say*
> "I think I see Batman!"

*But if this is the Penguin's Hideout level, then say*
> "Kwak! Kwak! I think I see Batman!"

*But if this is the Penguin's Hideout and the Penguin is apprehended, say*
> "That's Batman! Get him for what he did to the boss!"

*But if this is the Penguin's Hideout and there's no friends around, say*
> "Oh no Batman! Please don't hurt me!"

*But if the player is Catwoman, say*
> "It's the kitty kat!"

*But if the player is Catwoman and Poison Ivy is alive, say*
> "Hey cat, the boss wants a word with you"  ….

It's also natural to think of dialog as a system of general rules superceded by exceptions for particular circumstances. If a thug sees Batman, he says "hey look, Batman", *unless* he's a Penguin thug, *unless* he's a Penguin thug and the Penguin is arrested, *unless* he's alone, *unless* etc etc etc. This is a really comfortable way to think about behavior – as a hierachy of increasingly specific exceptions sitting on top of a general baseline.

# Criterion

- <u>A function that tests a fact</u>.
  - *eg* a key->func() pair that "matches" or "fails" a fact.
  - The predicate may be an equality, a range, or a function.
- { Query["who"] == "bill" }
- { Query["hitpoints"] > 30 && Query["hitpoints"] < 60 }
- { IsPrimeNumber( Query["nearbyEnemies"] ) == true }

If a "fact" is a single piece of state about the world, then a "criterion" is a single function that tests a fact for truth. Like "The speaker is Bill" or "hitpoints are between 30 and 60." I'm using "function" here in the computer science sense of some arbitrary conditional the returns true or false on a particular fact. You *could* use an actual function pointer if you really wanted to, but it's hardly ever necessary; typically we represent all of our criteria as numerical comparisons to make them more efficient. (That's later in the talk.)

# Rule

- **<u>A tuple of criteria to match against a query.</u>**
  - If all criterion are true, the rule "matches"
  - If any criterion has no matching fact in the query, it rejects.
- Many rules may match a query, so:
  - A scoring function (to pick specific rules over general ones).
  - We use a simple one: # of criteria matched.

A rule is a a tuple of criteria that all have to be true. If one is false, or one mentions a fact not in the query, then the rule is considered to reject.

Many rules may match a query. If you have a very specific "oh look a zombie is on the merry-go-round next to the ice cream machine" rule, then the "oh look a zombie" general rule will probably match also. So you need a scoring function to pick specific rules over general ones. You can write that lots of different ways; have different weights for criteria and so on. The scoring function that worked best for us was the simplest one imaginable – the number of criteria in a rule. The more criteria a rule has, the more specific it is.

# Rule Matching

Query

`{ who: nick, concept: onHit, curMap:circus, health: 0.66, nearAllies: 2, hitBy: zombieclown }`

PASS Rule 1: `{ who = nick, concept = onHit }` → *"ouch!"*

FAIL Rule 2: `{ who = nick, concept = onReload }` → *"changing clips!"*

FAIL Rule 3: `{ who = nick, concept = onHit, health < 0.3 }` → *"aaargh I'm dying!"*

PASS Rule 4: `{ who = nick, concept = onHit, nearAllies > 1 }` → *"ow help!"*

PASS Rule 5: `{ who = nick, concept = onHit, curMap = circus }` → *"This circus sucks!"*

PASS Rule 6: `{ who = nick, concept = onHit, hitBy = zombieclown }` → *"Stupid clown!"*

PASS Rule 7: `{ who = nick, concept = onHit, hitBy = zombieclown, curMap = circus}`
→ *"I hate circus clowns!"*
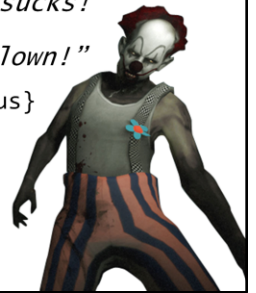
Here's a simple example of how you might match a query against some rules. You can see the facts in the query. The first rule matches, because both of its criteria are true. The second one fails because the "concept' criterion is wrong. The third one fails because the "health" criterion is wrong. The rest of the rules all have additional, more specific criteria, which match as well.

# Rule Matching

Query

`{ who: nick, concept: onHit, curMap:circus, health: 0.66, nearAllies: 2, hitBy: zombieclown }`

Rule 1: { who = nick, concept = onHit } → *"ouch!"*   *Score*: 2

Rule 4: { who = nick, concept = onHit, nearAllies > 1 } → *"ow help!"*   *Score*: 3
Rule 5: { who = nick, concept = onHit, curMap = circus } → *"This circus sucks!"*   *Score*: 3
Rule 6: { who = nick, concept = onHit, hitBy = zombieclown } → *"Stupid clown!"*   *Score*: 3
Rule 7: { who = nick, concept = onHit, hitBy = zombieclown, curMap = circus}   *Score*: 4
        → *"I hate circus clowns!"*

So now we score the rules that passed. The simplest way is just to count the number of matching criteria.
Rule 1 has two criteria, it's the general case, scores 2.
Rule 4 has more, it's more specific, so scores 3. It'll always play in preference to the other when available.
Rule 5 and 6 also match other specific criteria, scoring 3. They're all appropriate so you can choose randomly between them for variety.
But rule 7 has more criteria than the rest. It scores higher, so is the most specific line, the one that plays.

# Response

- <u>The payload of a rule</u>
- Can be anything
- In our speech system, lists of scripts combining anim + lipsync + sound
- Can contain intelligent logic also

```
Response PlayerLedgeHangStartNamVet
{
        scene "scenes/NamVet/LedgeHangStart01.vcd"
//I need a hand up!
        scene "scenes/NamVet/LedgeHangStart02.vcd"
//Somembody pull me up!
        scene "scenes/NamVet/LedgeHangStart03.vcd"
//Godammit, I'm hangin' off a ledge over here.
        scene "scenes/NamVet/LedgeHangStart04.vcd"
//Somebody needs to get over here and pull me up!
        scene "scenes/NamVet/LedgeHangStart05.vcd"
//Hey, people, I'm hangin' off a ledge over here!
        scene "scenes/NamVet/LedgeHangStart06.vcd"
//Somebody come pull me up!
        scene "scenes/NamVet/LedgeHangStart07.vcd"
//I'm gettin' too old for this hangin' shit.
        scene "scenes/NamVet/LedgeHangStart08.vcd"
//Somebody come help me up!
        scene "scenes/NamVet/LedgeHangStart09.vcd"
//Somebody come grab me up off this ledge.
}
```

The "response" is just whatever happens when a rule matches. You can have some intelligence here too. For example, we actually record a bunch of different variation for each line, put them in the same "response", and have the engine choose randomly between them when a rule matches; it's an easier way to have variety than creating a bunch of parallel rules. Or your "response" could be code, or executable script, or anything really.

# Writeback and followup

```
Response C3M2SafeRoomGambler
{
scene "scenes/Gambler/WorldC3M1B17.vcd"  then rochelle C3M2SafeRoom2d
         // Shit. This swamp is going to ruin my white suit.
scene "scenes/Gambler/WorldC3M2B03.vcd"  then gambler C3M2SafeRoomb2
         // These swamps don't agree with me.
scene "scenes/Gambler/WorldC3M2B05.vcd"
         // I am not dying in this goddamn swamp.
}


Rule C3M2SafeRoomGambler
{
criteria ConceptTalkIdle Joined3 IsGambler IsNotSaidC3M2SafeRoom ismapc3m2_swamp IsInStartArea
ApplyFacts   "SaidC3M2SafeRoom:1:0,Talk:1:3.088"
Response C3M2SafeRoomGambler
}
```

So far what we have is an elaborate system of conditional choice – basically a rearranged if/else and switch mechanism. To make the system capable of memory and conversation, we need more. Let's take a look at how  a particular rule works, in detail. In this case, one of the conversations that play at the beginning of a mission.

# Writeback and followup
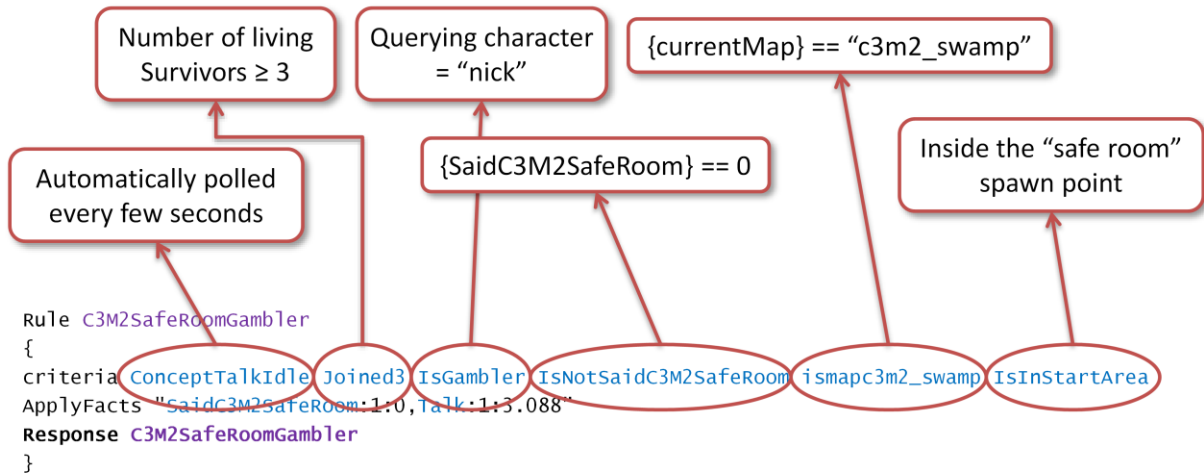


Each character in game polls itself every few seconds to see if it has any "I'm idle" dialog it wants to play. That's the "TalkIdle" concept. This rule will match that concept if some other criteria are met:

- Three survivors are present
- The speaker is "Nick", the "gambler"
- This line hasn't been said already
- We are in the swamp map
- We're in the start area.

# Writeback and followup

*Expiration time*

| Name | "gambler" | Ø |
|---|---|---|
| Zombies_killed | 112 | Ø |
| Molotovs_thrown | 4 | Ø |
| SaidC3M2SafeRoom | 1 | Ø |
| talking | 1 | now + 3.088" |

Stored context table

```
Rule C3M2SafeRoomGambler
{
criteria ConceptTalkIdle Joined3 IsGambler IsNotSaidC3M2SafeRoom ismapc3m2_swamp IsInStartArea
ApplyFacts   "SaidC3M2SafeRoom:1:0, Talk:1:3.088"
Response C3M2SafeRoomGambler
}
```

When this rule matches, it'll write a couple of facts back to nick's memory: in this case, that the line has been played, and that Nick is speaking for the next few seconds. The latter is an example to show that you can have automatic expiration times on a particular fact, if you want to prevent two successive bits of a running gag from being played too close together.

# Writeback and followup

```
Response C3M2SafeRoomGambler
{
scene "scenes/Gambler/WorldC3M1B17.vcd"  then rochelle C3M2SafeRoom2d
        // Shit. This swamp is going to ruin my white suit.
scene "scenes/Gambler/WorldC3M2B03.vcd"  then gambler C3M2SafeRoomb2
        // These swamps don't agree with me.
scene "scenes/Gambler/WorldC3M2B05.vcd"
        // I am not dying in this goddamn swamp.
}


Rule C3M2SafeRoomGambler
{
criteria ConceptTalkIdle Joined3 IsGambler IsNotSaidC3M2SafeRoom ismapc3m2_swamp IsInStartArea
ApplyFacts    "SaidC3M2SafeRoom:1:0,Talk:1:3.088"
Response C3M2SafeRoomGambler
}
```

Now look at those "then" clauses. What they mean is: once the line has finished, automatically trigger \*another\* concept (specified there) to the specified character. "Then rochelle C3M2SafeRoom2d" means that a "C3M2SafeRoom2d" concept is sent to Rochelle (the TV producer) after Nick finishes saying his line. She in turn will do a lookup in her rule database and find if there is a reply she wants to say.

# Writeback and followup

```
Response C3M2SafeRoomGambler
{
scene "scenes/Gambler/WorldC3M1B17.vcd"  then rochelle C3M2SafeRoom2d
                                                                    suit,

Response C3M2SafeRoom3dGambler
{
scene "scenes/Gambler/WorldC3M1B18.vcd"
//Brains come out. Swamp water doesn't. Don't ask me how
I know that.                                     gambler C3M2SafeRoom3d
}                                                t? That one?
Rule C3M2SafeRoom3dGambler
{
criteria ConceptC3M2SafeRoom3d IsGambler
Response C3M2SafeRoom3dGambler
}
```

Rochelle does have a reply. She says it; that in turn has another "then" followup that dispatches back to Nick, who has a reply of his own, and so on.

# Writeback and followup

```
Response C3M2SafeRoomGambler
{
scene "scenes/Gambler/WorldC3M1B17.vcd"  then rochelle C3M2SafeRoom2d
        // Shit. This swamp is going to ruin my white suit.
scene "scenes/Gambler/WorldC3M2B03.vcd"  then gambler C3M2SafeRoomb2
        // These swamps don't agree with me.

Response C3M2SafeRoomb2Gambler
{
scene "scenes/Gambler/WorldC3M2B01.vcd"  then mechanic C3M2SafeRoomb3
        //This swamp is just a cesspool for disease.
scene "scenes/Gambler/WorldC3M2B02.vcd"  //I can feel my feet growing fungus.
}
            Response C3M2SafeRoomb3Mechanic
            {
                    scene "scenes/Mechanic/WorldC3M1B01.vcd"  //Nick does have a point.
            }
```

Or maybe the other lines gets chosen randomly. Maybe instead of dispatching the line to Rochelle he dispatches it to himself. Then he has another couple of lines; "I can feel my feet growing fungus", or "This swamp is just a cesspool for disease." The latter one has *another* followup to Ellis. Each time the game plays out, it'll randomly select a different change, a different experience, without needing any special programmer code at all. The writers can build it all themselves, quite easily.

This approach also means you query the followup line when the callback happens, not when the first character starts to speak. The situation may have changed during the time it took the first line to be said.

# Self-branching conversation
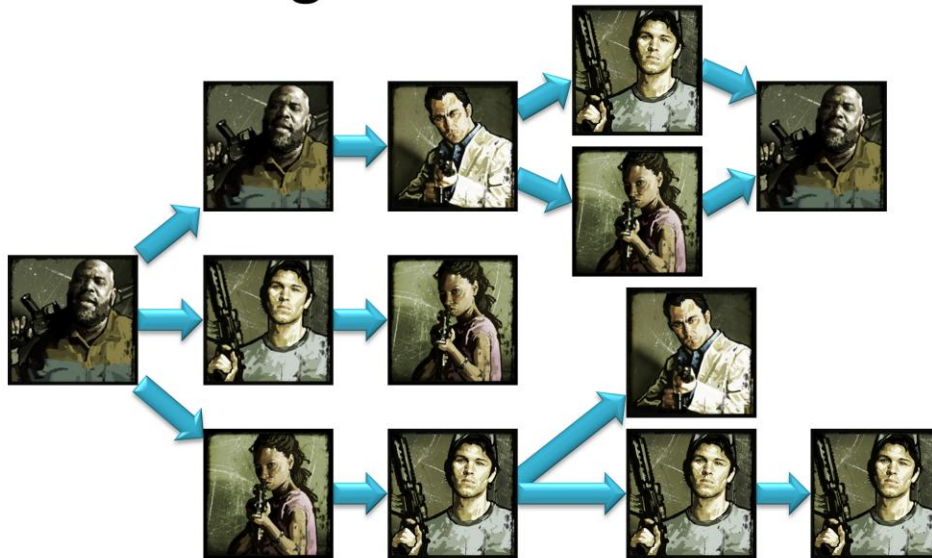


This is a really cheap way to get this branching conversation effect that we use to create variety. Just by adding a couple of rules and recording some voice, you can add new outcomes, or merge paths back together. You can create a lot of possible paths really cheaply.

# "then any" rules

- Dispatches followup to *all* characters within earshot
- Picks the highest-scoring rule from all of them
- Most specific reply wins

```
Response GoingToDieGambler
{
  scene "scenes/Gambler/GoingToDie01.vcd"  then any ally_dying  //I really screwed the pooch back there.
  scene "scenes/Gambler/GoingToDie02.vcd"  then any ally_dying  //I gotta take better care of myself.
  scene "scenes/Gambler/GoingToDie03.vcd"  then any ally_dying  //Not dead yet, but not exactly healthy.
  scene "scenes/Gambler/GoingToDie05.vcd"  then any ally_dying  //If I go, you guys gonna miss me.
  scene "scenes/Gambler/GoingToDie26.vcd"  then any ally_dying  //I am not dying in the middle of nowhere.
}
Rule GoingToDieCoachCoachGambler
{
  criteria ConceptPlayerGoingToDie IsNotSpeaking IsNotCoughing IsGambler IsAnyoneNear200
  Response GoingToDieGambler
}
```

In addition to dispatching a followup to a specific character, you can send one to all nearby characters within earshot simultaneously, to see if any of them have a reply; and of those which have the best reply.

best match = 0

best match = 2

best match = 6

```
Response GoingToDieGambler
{
  scene "scenes/Gambler/GoingToDie01.vcd"   then any ally_dying   //I really screwed the pooch back there.
  scene "scenes/Gambler/GoingToDie02.vcd"   then any ally_dying   //I gotta take better care of myself.
  scene "scenes/Gambler/GoingToDie03.vcd"   then any ally_dying   //Not dead yet, but not exactly healthy.
  scene "scenes/Gambler/GoingToDie05.vcd"   then any ally_dying   //If I go, you guys gonna miss me.
  scene "scenes/Gambler/GoingToDie26.vcd"   then any ally_dying   //I am not dying in the middle of nowhere.
}
Rule GoingToDieCoachCoachGambler
{
  criteria ConceptPlayerGoingToDie IsNotSpeaking IsNotCoughing IsGambler IsAnyoneNear200
  Response GoingToDieGambler
}
```

So the rule goes out to everyone. And maybe Rochelle doesn't have any line at all for this situation, so she has a match score of zero. Maybe Ellis has kind of a generic line, so he matches with score 2.

best match = 0

best match = 2

best match = 6

```
Response GoingToDieGambler
{
  scene "scenes/Gambler/GoingToDie01.vcd"    then any ally_dying  //I really screwed the pooch back there.
  scene "scenes/Gambler/GoingToDie02.vcd"    then any ally_dying  //I gotta take better care of myself.
  scene "scenes/Gambler/GoingToDie03.vcd"    then any ally_dying  //Not dead yet, but not exactly healthy.
  scene "scenes/Gambler/GoingToDie05.vcd"    then any ally_dying  //If I go, you guys gonna miss me.
  scene "scenes/Gambler/GoingToDie26.vcd"    then any ally_dying  //I am not dying in the middle of nowhere.
}
Rule GoingToDieCoachCoachGambler
{
  criteria ConceptPlayerGoingToDie IsNotSpeaking IsNotCoughing IsGambler IsAnyoneNear200
  Response GoingToDieGambler
}
```

But Coach, he likes to Coach people. He has specific lines for this situation. So, if he happens to be around, he'll match with the best score.

```
Response CoachHelps
{
        scene "scenes/Coach/GoingToDieR01.vcd"   //Come on now, put it all out there.
        scene "scenes/Coach/GoingToDieR03.vcd"   //That's it. Stay focused.
        scene "scenes/Coach/GoingToDieR07.vcd"   //Keep it up, come on, keep it up, keep it up. You're gonna make it.
        scene "scenes/Coach/GoingToDieR09.vcd"   //They put a hurtin on ya but ain't no thing.
        scene "scenes/Coach/GoingToDieR10.vcd"   //Come on now, put it behind ya, you good, you good.
}
Rule CoachHelps
{
        criteria Concept_ally_dying IsCoach IsNotSaidcoachcoaches IsHealthyHalf
        Response CoachHelpsCoach
}

Response CoachHelpsGambler
{
        scene "scenes/Coach/GoingToDieRGambler03.vcd"   //Nick, at least you dressed for a funeral.
}
Rule CoachHelpsGambler
{
        criteria Concept_ally_dying IsCoach IsNotSaidcoachcoaches IsHealthyHalf From_Gambler ChanceToFire30Percent
        Response CoachHelpsGamblerCoach
}

Response CoachHelpsMechanic
{
        scene "scenes/Coach/GoingToDieRMechanic01.vcd"   //Come on Ellis. Ya got it in ya.
        scene "scenes/Coach/GoingToDieRMechanic02.vcd"   //Come on youngin' if I can do it, you can do it.
}
Rule CoachHelpsMechanic
{
        criteria Concept_ally_dying IsCoach IsNotSaidcoachcoaches IsHealthyHalf From_Mechanic ChanceToFire30Percent
        Response CoachHelpsMechanicCoach
}
```

Not only does Coach tend to have the best match because he has specific "help my buddy out" lines, but he also has specific lines for each of the characters. They have an additional "if random number is less than 30" criteria so they don't get overplayed. Also, if somehow we added another survivor character for which he didn't have a specific line, it would automatically fall back to the general ones, and Coach would still have something to say.

So all of this creates character for Coach, and an interaction between the survivors! Coach coaches – that's who he is. And if you're hurting and he happens to be around and he's healthy, he'll try to coach you along. If he isn't around, maybe someone else has something to say. If not, then Nick just complains to himself. It's all automatic and writers can add additional special cases without needing to change any code.

# Each followup line is a new query!

- Cope with changing situations

♪ ?

```
Rule SawRidersPoster_Coach
{
    criteria ConceptSawRidersPoster IsNotInDanger IsCoach IsAnyoneNearby
    Response Coach_Likes_Riders // Hey, do you like the midnight riders?
    then Ellis RidersConversation1
}
```

:-)

```
Rule RidersConversation_Ellis
{
    criteria RidersConversation1 IsNotInDanger IsEllis IsAnyoneNearby
    Response Ellis_Likes_Riders // Yeah, they're awesome!

}
```

Query the followup line when the callback happens, not when the first character starts to speak. The situation may have changed during the time it took the first line to be said. For example, consider an interaction when coach talks about his favorite rock band, and Ellis agrees. Coach only sends a message to Ellis to look up his line *after* Coach's line is finished.

# Each followup line is a new query!

- Cope with changing situations

```
Rule SawRidersPoster_Coach
{
    criteria ConceptSawRidersPoster IsNotInDanger IsCoach IsAnyoneNearby
    Response Coach_Likes_Riders // Hey, do you like the midnight riders?
    then Ellis RidersConversation1
}


Rule RidersConversation_Ellis
{
    criteria RidersConversation1 IsNotInDanger IsEllis IsAnyoneNearby
    Response Ellis_Likes_Riders // Yeah, they're awesome!

}
```

That's to handle cases where, say, a zombie appears and starts chewing on Ellis while Coach is talking. In this case you do not want Ellis to continue blabbing about the Midnight Riders. You want him to interrupt the conversation and talk about something else.

Because Coach sends a message to Ellis at the *end* of Coach's line, Ellis does a lookup for a reply based on the context at exactly the moment he begins speaking. In that case, the IsNotInDanger criterion is no longer true; a zombie is nearby. So the conversation self-terminates because the criteria for its existence are no longer true. You don't need any kind of explicit interruption mechanism.

# Corrolaries

- **The followup line is selected *after* the first character speaks, *when* the second one replies**
  - To allow rules to adapt to changed state
- There's no "cutscene" entity that grabs both characters.
  - Just successive bits of dialog and statekeeping
- Cut long speeches up into little self-followup pieces

That gets you out of having to build explicit "conversation" entities and glue down both characters while they're speaking and have a means of handling interruptions, etc.

It's also worthwhile to cut up long monologues by a single character into short pieces, where each line sends a "followup" back to the speaker to trigger the next. That's a simple way to bail out of long speeches if something happens, or allow writers to create additional conversation branches where another character actually breaks in on the speech if they happen to be present.

# Code as Content

- Replace programmer work in a uniformly managable way
- Fewer dependencies
- Faster (or absent!) compile step
- Writers can work simultaneously on different characters
- Hot swapping & dynamic reloading

This blurs the line between code and content, with a number of salutary effects.

Humorous video.
Video available via
http://assemblyrequired.crashworks.org/gdc2012-dynamic-dialog/

# The Machine-Writer Interface
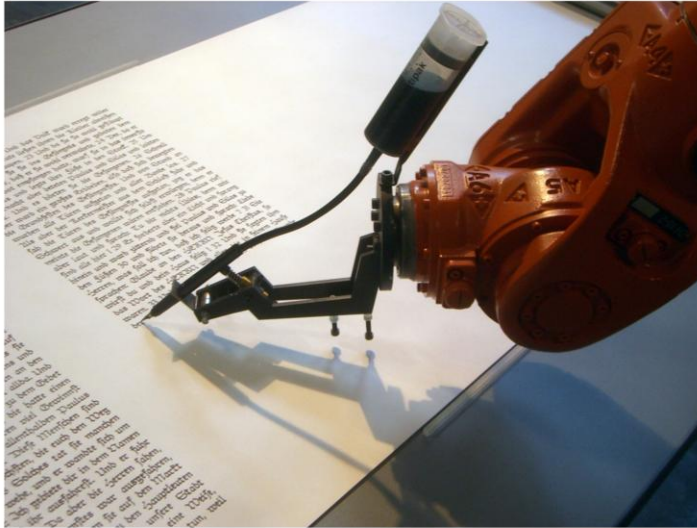


Now let's talk a bit about how to make a system that is comfortable and powerful for writers to work in.

Photo credit: bios [bible] robot by Matthias Gommel, Martina Haitz, Jan Zappe

# Prior art:
# Inform 7
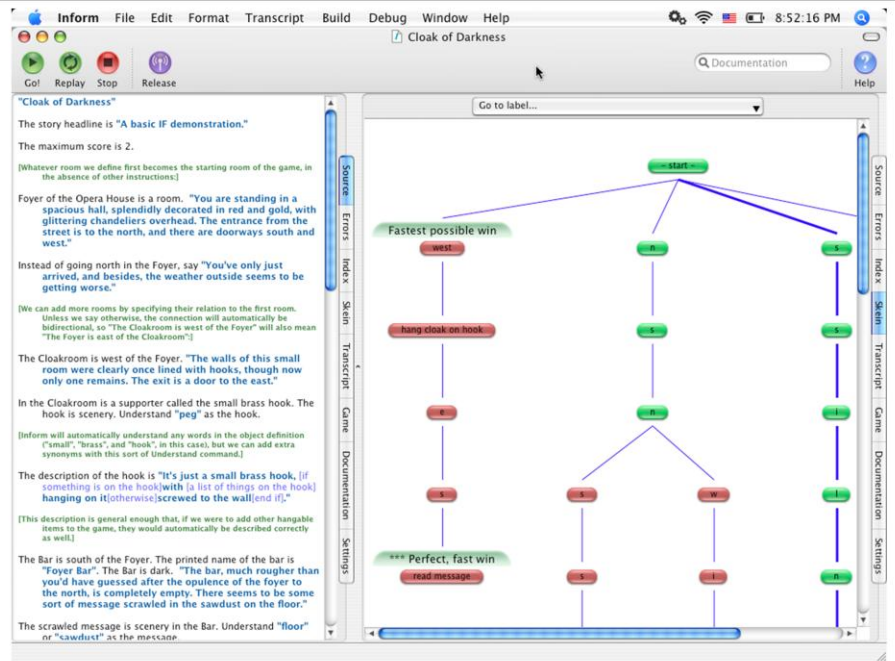
In fact, what better place to see how writers make code than interactive fiction? The Inform 7 language/system, used for making text adventures and interactive fiction, has had this notion of rulebooks cascading from simple to general cases for years.

# Prior art: Inform 7

*Feline Behavior* source code from the Inform documentation, http://bit.ly/yKCbCM

The Kitchen is a room. The cat is an animal in the Kitchen. In the Kitchen is a bowl, a ball of wool, a newspaper. The bowl contains a quantity of cream.
The player carries a closed openable container called a bag. The bag contains catnip.

The cat behavior rules is a rulebook producing an object.

A cat behavior rule when the cat can touch the catnip:
　say "The cat frolics with the catnip until nothing remains of it.";
　rule succeeds with result catnip.

A cat behavior rule when the cat can touch the cream:
　say "The cat laps up the cream.";
　rule succeeds with result cream.

A cat behavior rule when the cat can touch the ball of wool:
　say "The cat makes the ball of wool into a useless tangle.";
　rule succeeds with result ball.

A cat behavior rule when the cat can touch the newspaper:
　say "The cat bats playfully at the newspaper until all the nasty boring articles are destroyed.";
　rule succeeds with result newspaper.

A cat behavior rule:
　say "The cat looks miffed at the lack of ready entertainment, and glares at you with yellow eyes as though wondering whether your pants leg is good for claw-sharpening.";
　rule fails.

Every turn:
　let the destroyed object be the object produced by the cat behavior rules;
　if the destroyed object is not nothing:
　　remove the destroyed object from play;
　　say "[line break]Good thing you have no use for [the destroyed object] yourself.[paragraph break]".

This is actual Inform source code. It looks like English, but it's actually a regular computer-parsable grammar. Inform has this very notion of rulebooks, where specific cases override general ones based on circumstances around them. As you can see this idea is very straightforward to represent in a natural way, and its inclusion in Inform suggests how applicable it is to the task of creating dialog and narrative.

# Smart Writers Define Their Own Workflow
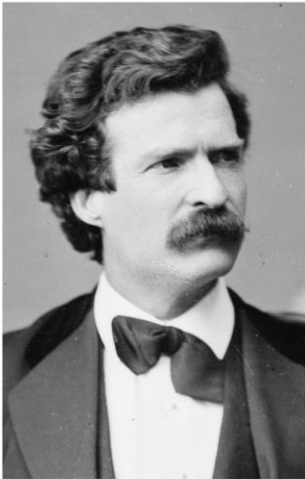


*Mark Twain*

*Sholes-Glidden typewriter*

Smart writers know what works for them. And your writers are smart. If they were not smart, they would not be writers.

Programmers shouldn't force a workflow on writers. It's important that writers be able to iterate as freely and quickly as possible, and they have to be comfortable for that to happen. Programmers often have preconceptions about how writers think, when many writers are perfectly able to define how they'd like their tools to work. Design tools for *your* writers on *your* project, rather than some abstract idea of writers. Ask *your* writers what they want.

This is a very early typewriter manufactured by the Sholes-Glidden company (later Remington), incidentally also the one that introduced the QWERTY keyboard. Mark Twain was one of the first writers to use one, because he loved gadgets. He didn't like this gadget, though; it didn't work well for him, and was unreliable. But he was perfectly capable of defining his own comfortable work environment better than Remington could.

Photos from Wikipedia.

# .txt files

- The actual asset the engine loads
- Simple recursive-descent grammar
- Parsed at load time (for quick reloading and iteration)
- Human readable but also easy to auto-generate

```
Rule MeleeDareCombatHeavy
{
            criteria ConceptPlayerBattleCry IsWeaponMelee IsHeavy IsCrosshairEnemy
            ApplyContext "IsDaring:1:5"
            Response MeleeDareCombatHeavy
}
Rule PlayerShinyCryHeavy
{
            criteria ConceptPlayerBattleCry 30PercentChance IsWeaponPrimary IsHeavy
WeaponIsNotVanillaPrimary WeaponIsNotTaggedMinigun
            Response PlayerShinyCryHeavy
}

Rule PlayerShinyWindupHeavy
{
            criteria ConceptWindMinigun IsWeaponPrimary IsHeavy
WeaponIsNotVanillaPrimary WeaponIsNotTaggedMinigun WeaponIsNotTomislav 5PercentChance
HeavyNotShinySpeech HeavyNotkillSpeech
            ApplyContext "HeavyShinySpeech:1:300"
            Response PlayerShinyCryHeavy
}

// Custom response battle cry against an Engineer.
Response PlayerTauntCryHeavy
{
            scene "scenes/player/heavy/low/332.vcd"
            scene "scenes/player/heavy/low/337.vcd"
}
Rule PlayerTauntCryHeavy
{
            criteria ConceptPlayerBattleCry 75PercentChance IsHeavy IsOnEngineer
IsCrosshairEnemy
            Response PlayerTauntCryHeavy
}

// Custom response for taunt against non-Heavies
Response PlayerTauntGunHeavy
{
            scene "scenes/player/heavy/low/329.vcd"
            scene "scenes/player/heavy/low/333.vcd"
            scene "scenes/player/heavy/low/335.vcd"
}
Rule PlayerTauntGunHeavy
{
            criterion ConceptPlayerBattleCry 75PercentChance IsHeavy IsNotOnHeavy
IsCrosshairEnemy NotGunTauntHeavy IsNotWeaponMelee
            ApplyContext "GunTauntHeavy:1:10"
            Response PlayerTauntGunHeavy
}
//End custom
```

One workflow is to simply write the script file that the engine loads directly. Ours has a straightforward recursive-descent grammar that is easy to parse and generate mechanically. A programmer can simply write the script in this format. I guess. It's not very convenient.

# Generating script from easier tools

*When seeing the player, say*
  "Look! It's Batman!"

*But if the player is in shadow, then say*
  "I think I see Batman!"

*But if this is the Penguin's Hideout level, then say*
  "Kwak! Kwak! I think I see Batman!"

*But if this is the Penguin's Hideout and the Penguin is apprehended, say*
  "That's Batman! Get him for what he did to the boss!"

```
Rule ThugSeeBatman
{
  concept OnSeePlayer
  criteria IsThug
  response Thug_LookItsBatman
}
```

Or, you can come up with a simpler specification language for writers to work in, and then cook that into script files. For example, consider the handwavey *Batman* example I showed earlier. Although these rules are specified informally, you can see how each of them could be mechanically transformed into a formal spec.

# Generating script from easier tools

*When seeing the player, say*
   "Look! It's Batman!"

*But if the player is in shadow, then say*
   "I think I see Batman!"

*But if this is the Penguin's Hideout level, then say*
   "Kwak! Kwak! I think I see Batman!"

*But if this is the Penguin's Hideout and the Penguin is apprehended, say*
   "That's Batman! Get him for what he did to the boss!"

```
Rule ThugSeeBatman_Shadow
{
    concept OnSeePlayer
    criteria IsThug  PlayerVisibility<0.5
    response Thug_SeeBatmanInDark
}
```

Or, you can come up with a simpler specification language for writers to work in, and then cook that into script files. For example, consider the handwavey *Batman* example I showed earlier. Although these rules are specified informally, you can see how each of them could be mechanically transformed into a formal spec.

# Generating script from easier tools

*When seeing the player, say*
   "Look! It's Batman!"

*But if the player is in shadow, then say*
   "I think I see Batman!"

*But if this is the Penguin's Hideout level, then say*
   "Kwak! Kwak! I think I see Batman!"

*But if this is the Penguin's Hideout and the Penguin is apprehended, say*
   "That's Batman! Get him for what he did to the boss!"

```
Rule PenguinThugSeeBatman
{
    concept OnSeePlayer
    criteria IsPenguinThug
    response PenguinThug_SeeBatman
}
```

Or, you can come up with a simpler specification language for writers to work in, and then cook that into script files. For example, consider the handwavey *Batman* example I showed earlier. Although these rules are specified informally, you can see how each of them could be mechanically transformed into a formal spec.

# Generating script from easier tools

*When seeing the player, say*
   "Look! It's Batman!"

*But if the player is in shadow, then say*
   "I think I see Batman!"

*But if this is the Penguin's Hideout level, then say*
   "Kwak! Kwak! I think I see Batman!"

*But if this is the Penguin's Hideout and the Penguin is apprehended, say*
   "That's Batman! Get him for what he did to the boss!"

```
Rule PenguinThugSeeBatman_Act4
{
    concept OnSeePlayer
    criteria IsPenguinThug
            IsPenguinJailed
    response PenguinThug_SeeBatman_NoBoss
}
```

Or, you can come up with a simpler specification language for writers to work in, and then cook that into script files. For example, consider the handwavey *Batman* example I showed earlier. Although these rules are specified informally, you can see how each of them could be mechanically transformed into a formal spec.

# Prior art: Inform 7

The Kitchen is a room. The cat is an animal in the Kitchen. In the Kitchen is a bowl, a ball of wool, a newspaper. The bowl contains a quantity of cream.
The player carries a closed openable container called a bag. The bag contains catnip.

The cat behavior rules is a rulebook producing an object.

A cat behavior rule when the cat can touch the catnip:
　say "The cat frolics with the catnip until nothing remains of it.";
　rule succeeds with result catnip.

A cat behavior rule when the cat can touch the cream:
　say "The cat laps up the cream.";
　rule succeeds with result cream.

A cat behavior rule when the cat can touch the ball of wool:
　say "The cat makes the ball of wool into a useless tangle.";
　rule succeeds with result ball.

A cat behavior rule when the cat can touch the newspaper:
　say "The cat bats playfully at the newspaper until all the nasty boring articles are destroyed.";
　rule succeeds with result newspaper.

A cat behavior rule:
　say "The cat looks miffed at the lack of ready entertainment, and glares at you with yellow eyes as though wondering whether your pants leg is good for claw-sharpening.";
　rule fails.

Every turn:
　let the destroyed object be the object produced by the cat behavior rules;
　if the destroyed object is not nothing:
　　remove the destroyed object from play;
　　say "[line break]Good thing you have no use for [the destroyed object] yourself.[paragraph break]".

With Inform 7, you *can* specify these rules formally, even if they look like English. So, since this is a parsable computer grammar…

92

# Generating script from easier tools

The cat behavior rules is a rulebook producing an object.

A cat behavior rule when the cat can touch the catnip:
  say "The cat frolics with the catnip until nothing remains of it.";

A cat behavior rule when the cat can touch the cream:
  say "The cat laps up the cream.";

A cat behavior rule when the cat can touch the ball of wool:
  say "The cat makes the ball of wool into a useless tangle.";

```
Rule Cat_001
{
    concept CatBehavior
    criteria IsCat  IsTouchable($CATNIP)
    response R_CatFrolicsNip
}
```

```
Rule Cat_002
{
    concept CatBehavior
    criteria IsCat  IsTouchable($WOOL)
    response R_CatTanglesWool
}
```

… it too can be translated directly into the internal representation.

# Excel spreadsheet

| Response | npc_dota_hero_omniknight | Concept | Criteria |
|---|---|---|---|
| omni_rival_02 | Your congregation of squirrels and chipmunks cannot save you, Chen. | Kill | IsEnemyChen |
| omni_rival_03 | Doombringer, in you I have bested the inferno itself. | Kill | IsEnemyDoom_Bringer |
| omni_rival_04 | Skeleton King, your lordship is at an end. | Kill | IsEnemySkeleton_King |
| omni_rival_05 | Lifestealer, I will find what did this to you. | Kill | IsEnemyLife_Stealer |
| omni_rival_06 | Reality is no longer your plaything, Weaver. | Kill | IsEnemyWeaver |
| omni_rival_07 | The prophecies seem to have overestimated you, Bane. | Kill | IsEnemyBane |
| omni_rival_08 | Tinker, your science meddles where mortals should not. | Kill | IsEnemyTinker |
| omni_rival_09 | Dragon Knight, an impure body is an impure soul. | Kill | IsEnemyDragon_Knight |
| omni_rival_10 | The Omniscience would like a word with you, Zeus. | Kill | IsEnemyZeus |
| omni_rival_12 | Such power Warlock, is kept secret for a reason. | Kill | IsEnemyWarlock |

- Rules, comments, concepts, criteria entered as columns in a spreadsheet.
- Macro/tool exports the .xls into the response.txt format
- You can automatically export the .xls into the actor's callsheet for voice recording
- A convenient interchange format between tools

Dota's writer uses an Excel spreadsheet; and that's fine! Some writers like spreadsheets. There's a row for each bit of dialog, columns for the criteria, and we use a macro to export from this to the engine's format. An advantage of this system is that it keeps all of your information about voice in a common place; we can use the same .xls to track engine rules and data about the voice as it moves through casting, recording, and audio processing. We can export from this spreadsheet to both the engine's script and also the physical paper script that the actor takes into the recording booth.

# Visual tools



Editor Mockup.exe

Gambler_C3M2SafeRoom → then → rochelle_ C3M2SafeRoom2d, mechanic_ C3M2SafeRoom2d, gambler_ C3M2SafeRoom2g → mechanic_ C3M2SafeRoom3d
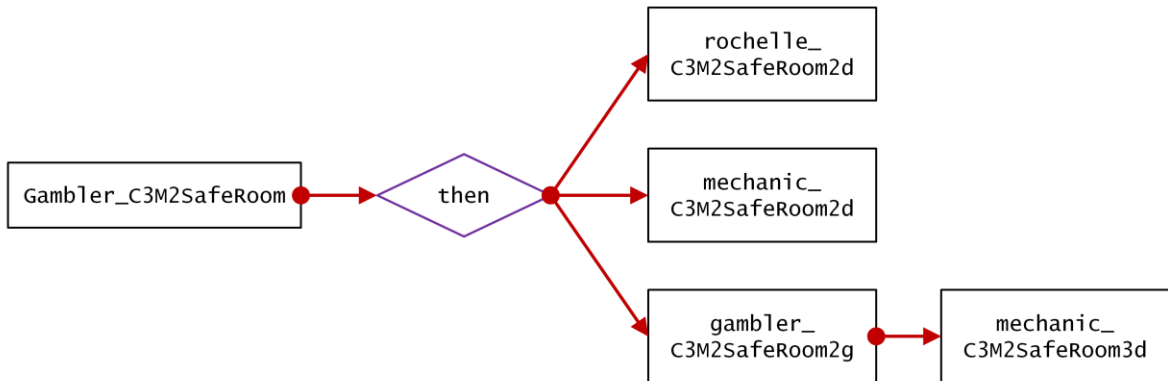
Or you can build a visual tool. I spent several weeks over one summer writing a gadget for writers to visualize their work and conversation flow – I haven't got a screenshot of it any more, but this is sort of what it looked like. But I fell into the trap of thinking about abstract "writers" rather than *my* writers. I sat down to write a tool and thought, "hm, what would writers like? Writers are creative people. Creative people like visual things. So what I need is a visual tool with drag and drop and little bubbles and…"

But I was wrong! I showed it to my writers and it never got used. It was too restrictive and too simplified for them. What the writers on Left4Dead and Portal really wanted was…
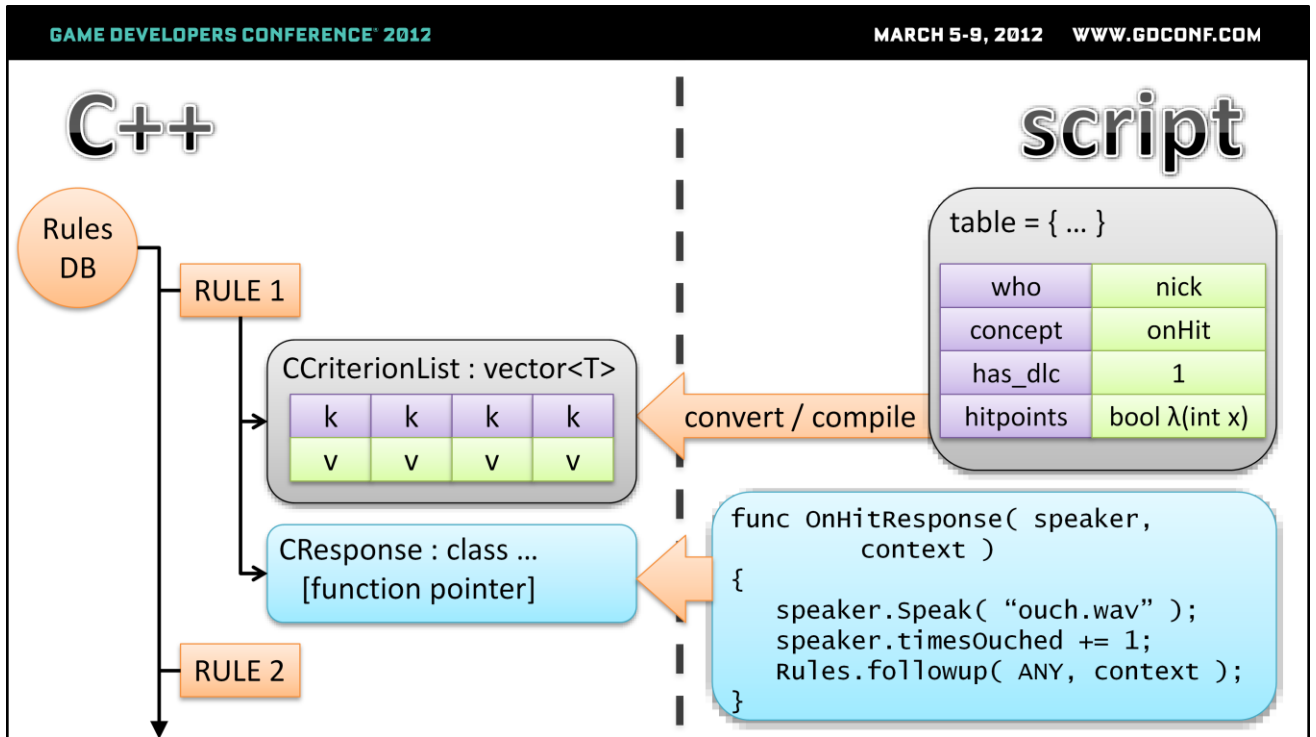
95

# FoxPro database (!)



…a FoxPro database.

Chet Faliszek and Erik Wolpaw were database administrators in a previous life. They're very comfortable with building databases, and were perfectly capable of building a FoxPro utility to manage all their work for them. And this worked *great* for them: they were able to corral enormous amounts of data, do batch-processing, write export scripts, store everything in one place. If I'd only asked them what *they* wanted, I could have spent my time writing them better FoxPro export tools or a frontend to some more usable database with the same features.

# As a Script Feature

- Implement a rules database on the C++ side
- Expose bindings to script
- Script tables convert neatly to queries, criteria lists
  - Criteria can be arbitrary functions (that map a fact's value to **true** or **false**)
- Rule "responses" can be arbitrary script objects
  - *Especially* functions!

You can also expose the rules database and its types as a feature in your script engine. Table-based script types map neatly to the notion of "facts", criteria lists, queries built as associative arrays; and your responses can be arbitrary script objects. By exposing bindings to your native-coded response engine, you can make it a script feature that's both convenient and performant.

# C++

**Rules DB**

**RULE 1**

CCriterionList : vector<T>

| k | k | k | k |
|---|---|---|---|
| v | v | v | v |

CResponse : class …
  [function pointer]

**RULE 2**

# script

table = { … }

| who | nick |
|-----|------|
| concept | onHit |
| has_dlc | 1 |
| hitpoints | bool λ(int x) |

**convert / compile**

```
func OnHitResponse( speaker,
          context )
{
    speaker.Speak( "ouch.wav" );
    speaker.timesOuched += 1;
    Rules.followup( ANY, context );
}
```

You can also expose the rules database and its types as a feature in your script engine. Table-based script types map neatly to the notion of "facts", criteria lists, queries built as associative arrays; and your responses can be arbitrary script objects. By exposing bindings to your native-coded response engine, you can make it a script feature  that's both convenient and performant.

# Debugging Tools

- Print any query / fact tuple to console
- Log queries
- Log sources of facts
  - "where did 'has_quest_x' get set from?"
- Log all matched rules
  - "why did this one score highest?"
- Log all *tested* rules, and which criteria passed/failed
- Dump current facts on any object (procedural and stored)
- Hot swap / edit & continue
- Asset validation – check that .wav files are there when scripts are loaded; check consistency; etc

Another important factor in usability is a rich set of debugging tools that can be used in-engine while the game is running. Make sure to write them!

Another humorous video.

Video available via
http://assemblyrequired.crashworks.org/gdc2012-dynamic-dialog/

# Nerd time

- Not a relational database
- A complex lookup function where each rule's key is a composite tuple of predicates
- The query is a tuple of values
- A rule "matches" if all predicates intersect with the query
  - (eg if the logical "AND" of all predicates called on the corresponding facts returns true)

*Principia Mathematica* (Isaac Newton)
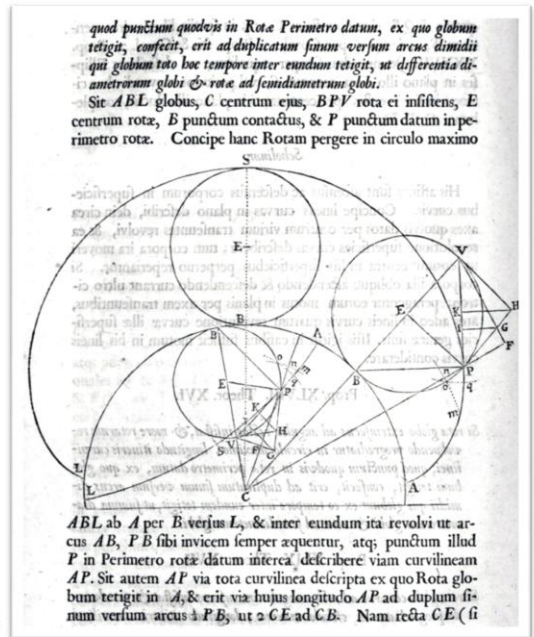
Now to some of the implementation details.
This is not a relational database. You cannot represent this as an SQL query. If it were a relational database, you would need a row for each rule and a column for every possible criterion to appear, meaning that most rules would have thousands of NULL columns for all the criteria they do not care about. This is neither efficient nor convenient.

In computer science terms what we have is a surjective lookup function. Each "rule" in the database is a key-value pair, where the value is the response and the key is an expected state of the world. In this case, the keys are not numbers or values, but complex predicate functions – in particular, a tuple of predicates, all of which must be true for the key to match. The predicate functions act upon the "query" object, which is a tuple of values. Another way to look at is that the predicates are global and look at the state of the world.

# The naïve implementation

- Query is a list of *eg* pair<string,variant>
  - aka "an associative array"
- Merge/catenate multiple fact dictionaries together to build up the query
  - aka "a union of associative arrays"
- For each rule:
  - For each criterion in the rule:
    - Look up the corresponding fact in the query
    - If missing or no match, reject rule
  - If all criteria match, add rule to "accept" list
- Return highest scoring rule
- For *r* rules, *c* facts in query, *d* criteria per rule:
  $O( r \times c \times d ) \approx O( n^3 )$

You can imagine the most straightforward way of doing this pretty easily. You start by adding together all of the sources of facts…

# Building a Query



| concept | "OnHit" |
|---------|---------|
| attacker | "hunter" |
| damage | 12.4 |

| hitpoints | 78 |
|-----------|-----|
| ammo | 43 |
| nearest_Ally | "coach" |
| nearest_ally_dist | 733.0 |
| current_weapon | "Axe" |
| … | … |

| times_healed | 4 |
|--------------|-----|
| suit_complaints | 2 |
| zombies_killed | 204 |
| times_used_axe | 111 |
| times_used_shotgun | 93 |

| map | "swamp 2" |
|-----|-----------|
| coach_alive | true |
| witches_killed | 7 |
| encountered_hillbilly_scene | false |
| total_zombies_killed | 1979 |
| … | … |

…into one giant associative array, by doing a merge operation (like adding dictionaries in Python)…

# The naïve implementation

- Query is a list of *eg* pair<string,variant>
  - aka "an associative array"
- Merge/catenate multiple fact dictionaries together to build up the query
  - aka "a union of associative arrays"
- For each rule:
  - For each criterion in the rule:
    - Look up the corresponding fact in the query
    - If missing or no match, reject rule
  - If all criteria match, add rule to "accept" list
- Return highest scoring rule
- For *r* rules, *c* facts in query, *d* criteria per rule: $O( r \times c \times d ) \approx O( n^3 )$
  = "You're fired!"

…and then doing the obvious thing.

In cubic time.

I pronounce cubic-time algorithms as "you're fired."

We can do better.

# Optimization #1: Sorted list linear walk

- Sort the list of criteria in each rule *and* the facts in the query.

**Q** | A : 8 | B : 6 | C : 7 | D : 5 | E : 3 | F : 0 | G : 9 |

**R** | A > 6 | B = 6 | E < 5 | G = 9 |

- If a criterion is missing from the query, reject.
- For r rules, c facts in query, d criteria per rule:
  $O( r \times d )$ or $O( r \times c )$
  $= O(n^2)$
  = "you're still fired."

First, simply sort the criteria and facts in each rule alphabetically. Then you can walk through them linearly, rather than having to search the query. Also, this makes it easy to early-reject when a rule has a criterion with no matching fact in the query.

# Opt. #2: Skip the merge step

*f*()

| A : 8 | D : 5 | E : 3 | F : 0 |
|-------|-------|-------|-------|

| B : 6 | G : 9 |
|-------|-------|

R

| A > 6 | B = 6 | E < 5 | G = 9 |
|-------|-------|-------|-------|
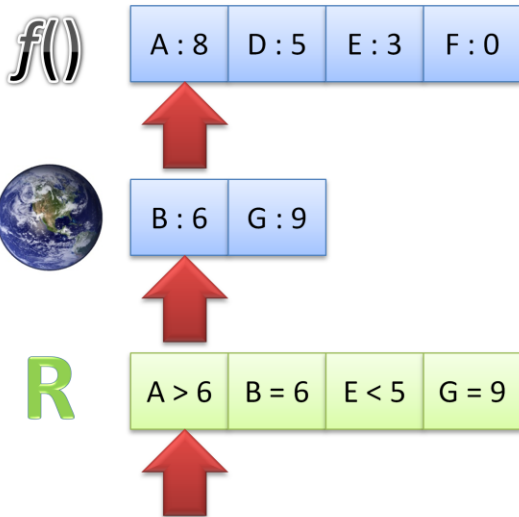
- No need to merge query + procedural + stored + world fact tuples together before sorting
- Store character and procedural facts in sorted arrays
- Move search pointers in parallel while matching criteria

Also, you don't need to actually merge the arrays. If you keep parallel pointers into each source of facts, then you can walk them individually and get mathematically the same result as actually merging the arrays, without having to actually perform the memcopy.

# Opt. #3: Hierarchical partition

- Pick a few keys that you include in *every* rule ( "concept" and "who" are good ones)
- Make a tuple of them and hash
- Partition the tables by that so you can quickly reject ones that don't match
- For rules divided into *p* partitions, you get O( *r/p* × *d* )

Next you can partition your rules. For example, if you know that every rule always has a "who" criterion identifying who is speaking,

# Opt. #3: Hierarchical partition

| rule | rule | rule | rule | rule | rule | rule | rule | rule | rule | rule | rule |
|------|------|------|------|------|------|------|------|------|------|------|------|
| crit | crit | crit | crit | crit | crit | crit | crit | crit | crit | crit | crit |
| crit | crit | crit | crit | crit | crit | crit | crit | crit | crit | crit | crit |
| crit | crit | crit | crit | crit | crit | crit | crit | crit | crit | crit | crit |

Then there is no need to search every rule in the database for lines pertaining only to Nick.

## Opt. #3: Hierarchical partition
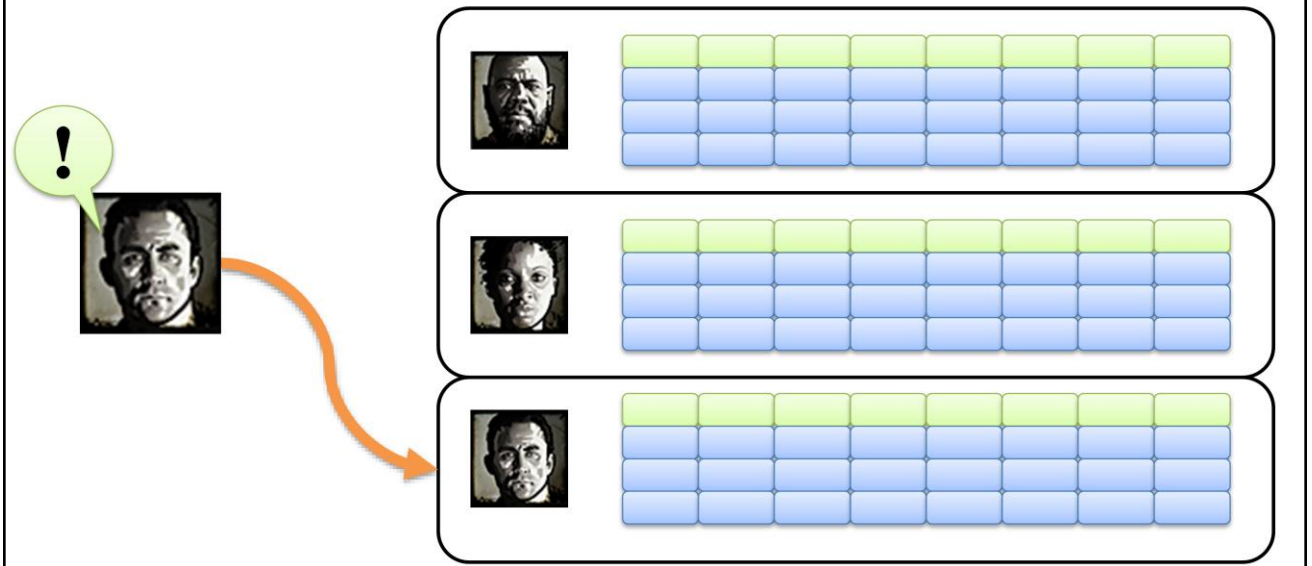
You can bucket your rules by speaker, get a constant-time lookup into the partition containing just Nick's rules, and then search just his lines.
You can also subdivide *that* partition further by concept, map, etc – anything else that you know to be a constant-value predicate present in each rule.

You can also take those predicates – like "who=nick ; concept=onreload ; map=swam" – concatenate, and hash them. That gives you a hash key you can use to bucket rules as finely as you like, rather than explicitly chopping them into partitions. That way you can specify arbitrarily many partitions based on how many keys you are hashing, so you can partition your rules as finely as you like. If you can get down to about fifty rules per bucket, and each rule has an average of eight criteria, you can do a lookup in less than a microsecond.

# Opt #4: Partitioning rules by region

- What about rules that only matter in one part of the world?
- And facts that are only relevant there too?



You can also explicitly partition rules and facts by region. Let's say you have a globetrotting European adventure. England and Spain have quests relevant only to those regions. When you're in England, you don't want to test all the rules that are relevant only to Spain quests; there is no chance that a line there will match.

# Individual databases per region/act/etc



So cut your rules up into individual databases by region. You'll always have the "global" rules which can play anywhere loaded (like "ouch" and "draw your sword!" and so on).

But rules in other regions can be put in their own databases. If you're in the King Arthur level, you don't even need to keep, say, the Italy rules in memory. Leave them on disk. Stream dialog rules in with the level data. Then when you search for lines in England, you can check the England and Global databases in parallel.

Photo of King Arthur from Monty Python and the Holy Grail
Illustrations of England and Italy from Wikimedia Commons.

# Building a Query



| concept | "OnHit" |
|---------|---------|
| attacker | "hunter" |
| damage | 12.4 |

| hitpoints | 78 |
|-----------|-----|
| ammo | 43 |
| nearest_Ally | "launce-lot" |
| nearest_ally_dist | 733.0 |
| current_weapon | "Axe" |
| ... | ... |

| Saxon_cities | 4 |
|--------------|-----|
| Bunnies_slain | 2 |
| IsMerry | true |

| current_map | "swamp 2" |
|-------------|-----------|
| emperor_alive | true |
| witches_killed | 7 |
| Quest5_complete | false |
| total_zombies_killed | 1979 |

Do the same thing with fact sources. Facts relevant only to England quests can be stored along with other England-specific data. You can merge in the England tables while running one of its quests, and dump the entire table of England facts from memory when in some other region.

# Opt #5: Sort rules by score or #criteria

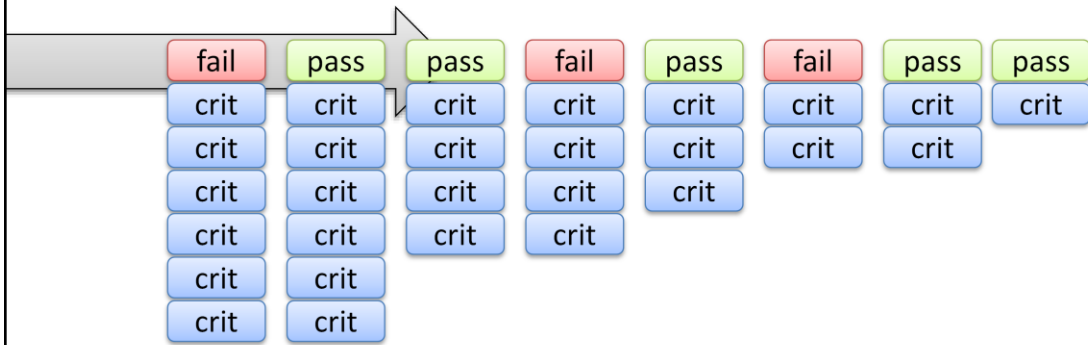| pass | fail | fail | pass | pass | pass | pass | fail |
|------|------|------|------|------|------|------|------|
| crit | crit | crit | crit | crit | crit | crit | crit |
| crit | crit | crit | crit |      | crit | crit | crit |
| crit | crit |      | crit |      | crit |      | crit |
|      | crit |      | crit |      | crit |      | crit |
|      | crit |      |      |      | crit |      |      |
|      | crit |      |      |      | crit |      |      |
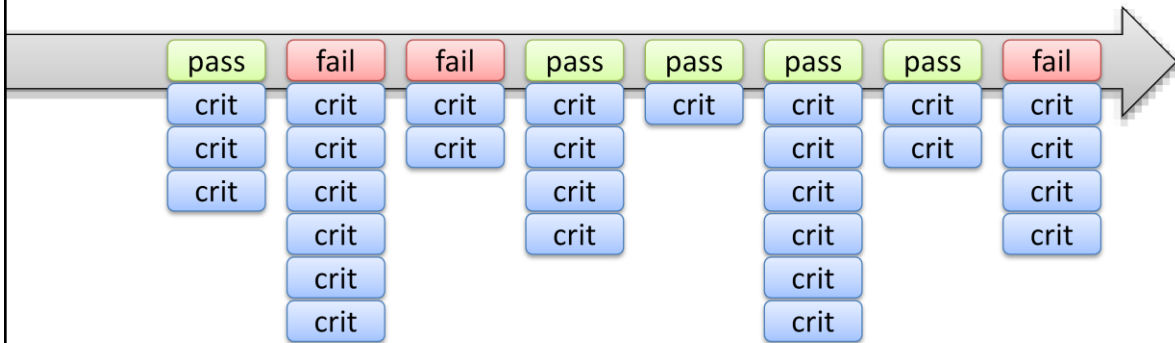
Next optimization: within each partition, search rules by decreasing score. If you match a six-criterion rule, there is no need to even test the five- and four-criteria rules; there's no way they'll be returned.

# Opt #5: Sort rules by score or #criteria

| fail | pass | pass | fail | pass | fail | pass | pass |
|------|------|------|------|------|------|------|------|
| crit | crit | crit | crit | crit | crit | crit | crit |
| crit | crit | crit | crit | crit | crit | crit | |
| crit | crit | crit | crit | crit | | | |
| crit | crit | crit | crit | | | | |
| crit | crit | | | | | | |
| crit | crit | | | | | | |

If you know a six-criterion rule has passed, there's no need to test the four-rule criteria. So sort by decreasing "score" and you don't need to test rules just to throw them out.

# Opt. #6: Speed up the linear phase

- The comparison of one fact vs criterion

- `"name" = "bob"`
- `hitpoints > 25 && hitpoints < 75`
- `numZombies < 3`

Next up you can accelerate the comparison of an individual criterion – eg "does name equal bob" and all those other building blocks.

# Opt. #6: Speed up the linear phase

Most comparisons can be represented as intervals on a number line.



Almost every criterion I have encountered can be represented as an interval on a number line.

Remember that IEEE754 supports comparing floating point numbers to infinity!

# Opt. #6: Speed up the linear phase

Most comparisons can be represented as intervals on a number line.

*"Use symbols, not strings!"*

$$name = \text{``bob''} = \#3478$$



-∞  ←-----------------●------------------→  +∞
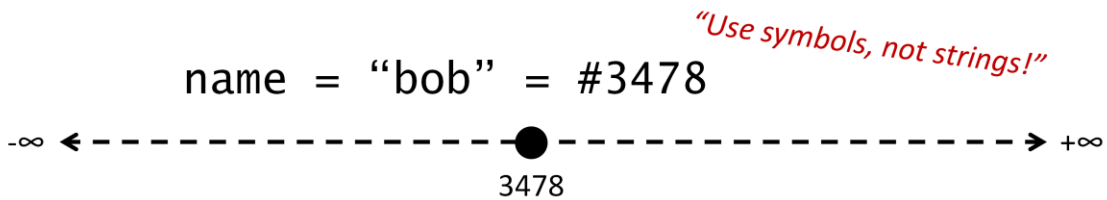
3478

$$name \geq 3478 \,\&\&\, name \leq 3478$$

Even string equality is an interval on a number line, if you use a symbol table or some other way of mapping strings to unique integers (as opposed to using **const char \*** like a noob).

You'll notice that even though I am asking == here, I actually use greater-than-or-equal AND less-than-or-equal to intersect to just "equal." This is so that every comparison can be performed using the exact same instruction stream.

# Opt. #6: Speed up the linear phase

So most criteria can be represented as a numerical function :

$$a \leq x \leq b$$

```cpp
struct CriterionStatic
{
  float fa;
  ComparisonType_t ctype;
  bool Compare( float x );
};
```

```cpp
bool CriterionStatic::Compare( float x )
{
  if ( ctype == EQUALS )
    return x == fa;
  else if ( ctype == GREATER_THAN )
    return x > fa;
  else if ( ctype == LESS_THAN_EQ )
    return x <= fa;
  // etc ad infinitum
}
```

It's possible to store a "comparison type" enum in each criterion and then switch or if-else between "equals", "greater", "greater-or-equal", "neq" etc comparisons between a parameter and a number. But this is a lot of additional branches.

# Opt. #6: Speed up the linear phase

So most criteria can be represented as a numerical function :

$$a \leq x \leq b$$

```
struct CriterionStatic
{
  float fa, fb;
  bool Compare( float x )
    { return x >= fa && fb >= x; }
};
```

*All* of those comparisons can be transformed into an a ≤ x ≤ b operation, or intersections thereof. Then you can represent every comparison as the same structure and use the exact same comparison code for each one. This reduces branch penalties drastically.

# Opt. #6: Speed up the linear phase

Reduce branch penalties by converting *all* comparisons to
a ≥ x ≥ b intervals.

In a discrete number line
  (like IEEE754 floating-point numbers),

$$x > a \longleftrightarrow x \geq a+\varepsilon$$

```cpp
struct CriterionStatic
{
  float fa, fb;
  bool Compare( float x )
    { return x >= fa && fb >= x; }
};
```

To learn all about comparing
floating-point numbers, see
Bruce Dawson's blog:

http://bit.ly/wQqozK

You can also do this with floating point numbers. In any discrete number system, you
can transform a strict greater-than comparison to a greater-or-equal comparison by
adding an epsilon. Epsilon does not mean "an arbitrarily small floating point number",
but has a specific definition in the context of comparing IEEE754 floats. Bruce
Dawson's blog has lots of great information about comparing floatpoint numbers
efficiently and the underlying details of their operation.

# Opt #7 - ∞· mad science land

- R ... rch problem ... ce
  -
  - ...es...
- G ...er that mea ... the
  m ... order
  - ...g a BSP tree
  - ...calculation s
- Principal Component Analysis to c ...
  lumped together in a partition
- Vector quantization, clustering analysis, math galore
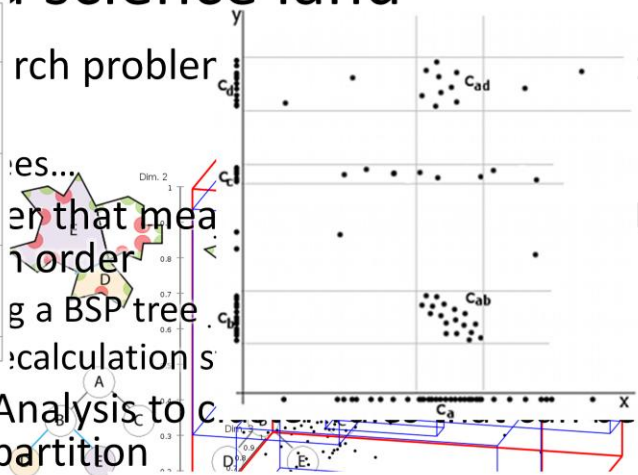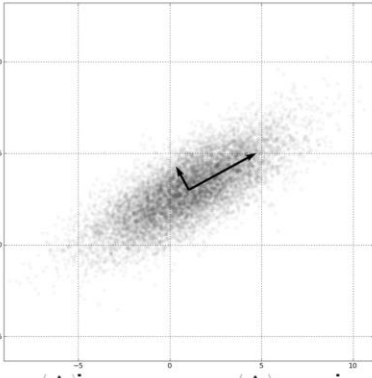
Then you can do all sorts of other clever optimizations – representing the intervals as subspaces of an n-dimensional space, partitioning rules by principal component analysis, using r-trees and x-trees and…
Don't.

121

# In practice…

- The spatial partitioning algorithm was a bridge too far.
- Hierarchical hashed partition is fine
- R-trees of *Q*-space are complicated and mind-bending
  - And they blow up the L2 cache too
- Opt #6 lets us query across 10000 rules in microseconds
- Can always do the R* thing if your data sets get Google-sized
- (see bibliography at end)

It's not worth it. It's a lot of extra complexity and in my experience not even faster; you end up blowing your cache more than you save time. If you just use the hierarchical partitioning mechanism, you can get your buckets down to a dozen or so rules apiece, and then finding the best rule in a bucket is less than a microsecond. The hierarchical technique is fast enough, and much simpler to code.

You can always go back to the crazy-land algorithms if you end up with enormous data sets; the interface to the system will remain the same, so you can optimize the back end ad lib.
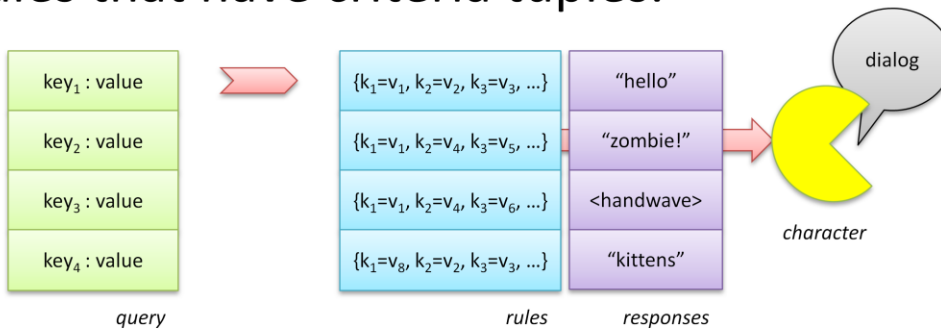
- Additive specialization = new quests and character types without modifying old code!



By the way, this makes modding and user-generated-content easier, because it's all additive. People can always *add* rules to the system without breaking old ones; and if you throw all the worldstate into each query, modders can add in new special cases for state that exists in the base product, but had no specific outcomes. So if a modder adds a new character class to the game, they can add in lines for the class, and even have old characters respond to it specifically, without needing to change the base product.

# The Essential Bits:

- A pattern-matching engine that does fuzzy search between a context tuple, and a set of rules that have criteria tuples.



| query | | rules | responses | character |
|-------|--|-------|-----------|-----------|
| $key_1$ : value | | $\{k_1=v_1, k_2=v_2, k_3=v_3, ...\}$ | "hello" | dialog |
| $key_2$ : value | | $\{k_1=v_1, k_2=v_4, k_3=v_5, ...\}$ | "zombie!" | |
| $key_3$ : value | | $\{k_1=v_1, k_2=v_4, k_3=v_6, ...\}$ | <handwave> | |
| $key_4$ : value | | $\{k_1=v_8, k_2=v_2, k_3=v_3, ...\}$ | "kittens" | |

So, a  summary. If you want this kind of rule-driven matching behavior, what you need is a pattern matching engine of some kind.

# The Essential Bits:

- Building context at runtime from
  - state about the world
  - state about the character
  - state about the event triggering the speech
    - (ie, speech type, presence of enemies, if "reloading" what kind of ammo, etc)

$f()$ **+**  **+** memory **+** 🌎

You want to build queries into your rule system by adding together as many facts about the world as possible; and you always want to throw all that state at the database each time, to enable writers to add new rules for new specific circumstances without requiring programmers to go and add additional data to the query.

# The Essential Bits:

- Followup rules: allow one line to trigger a query on
  - Another character
  - Any other character
  - All other characters



You need a way for one response to trigger a lookup on a different character when it has finished, to make conversations.
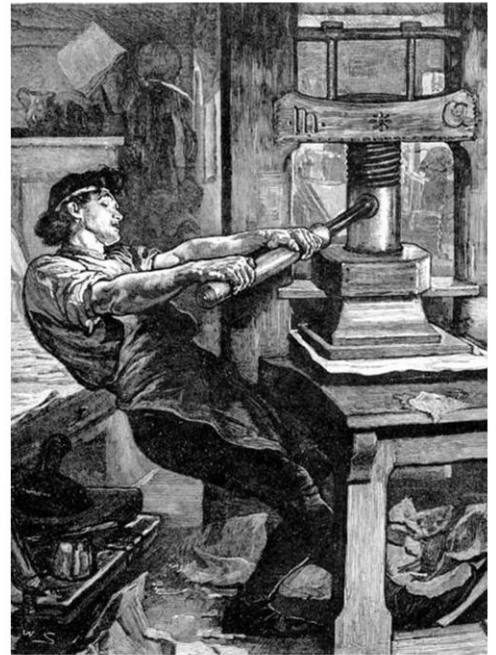
# The Essential Bits:

- Writeback of saved context to world or character
- Creates running gags, memory
- Enables real logic



You need a way for a matched response to write state back to the world, to create memory and turing-completeness.

# The Essential Bits:

- Convenient interface!
- Enable writers to do more work themselves.



And it has to be convenient for YOUR writers to work with! The whole point of this is to make a system that's comfortable, friendly, and intuitive for writers to work autonomously without having to wait on programmers. The more easily and quickly writers can iterate, the better they will write!

# Special Thanks

- Jeep Barnett
- Bruce Dawson
- Joe Demers
- Chet Faliszek
- David Kircher
- Lars Jensvold
- Scott Ludwig

- Sergiy Migdalisky
- Jason Mitchell
- Iestyn Bleasdale-Shepherd
- Erik Wolpaw
- Matt Wood
- And everyone at Valve! (even Doug)

# Questions?

Elan Ruskin
Twitter: @despair

Bibliography and slides at
http://bit.ly/yeHMST
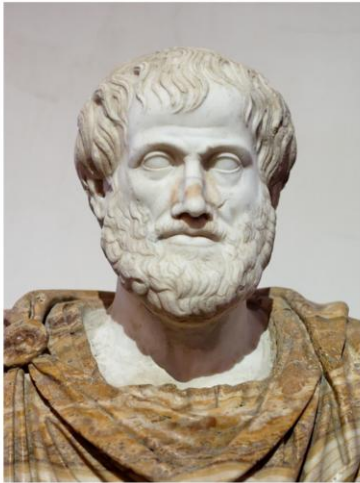(once I get home)



**bios [bible]** robot by Matthias Gommel, Martina Haitz, Jan Zappe
http://www.robotlab.de

# QUESTION AND ANSWER SLIDES

# A philosophical interlude



Ἀριστοτέλης / Aristotle

A philosophical interlude: when I say "Zoey remembers she got shot", is this meaningful or do I just mean "this creates the illusion that Zoey remembers she got shot." She doesn't "remember" anything, she is just choosing a different canned recorded line to play based on the state of a few variables.

# "Hamlet is upset about his dead father"
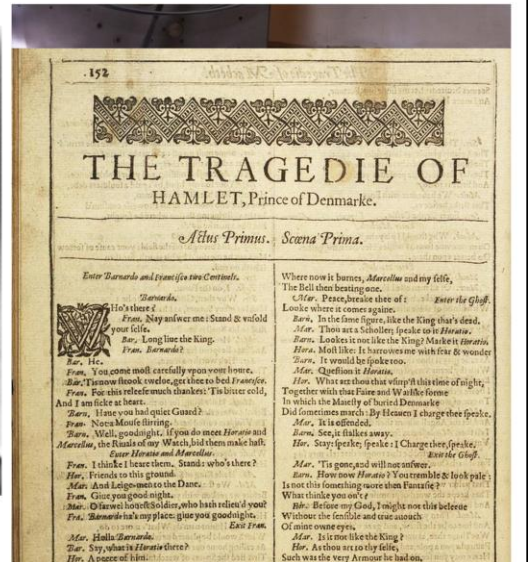
Well, what if I said "In that movie, Hamlet is upset about his dead father"?

I would actually mean "I saw a guy called Sir Laurence Olivier pretend to be Hamlet who is upset about his dead father." Actually what I really saw was a screen reflecting light projected through celluloid that on it had an image of Lawrence Olivier pretending to be Hamlet. But Hamlet is an imaginary person; what I really saw was a screen reflecting a picture of Laurence Olivier reading some words from a book written centuries earlier.

The point is that whether a character remembers or feels something is intrinsically a projection by the player, which is sustained by convincing writing and performance. If the object on screen acts and makes sounds like a convincing human would, we imbue it with human feelings. The quality of writing and simulation is what creates the suspension of disbelief.
Therefore it's important to make a system that enables writers to work comfortably and spontaneously.

> If the rule set is programmer defined, then you force writers to fill out a series of mad-libs, which is not going to generate quality content.
> Also, if the writers can't easily define new rules, then they won't spontaneously come up with ideas for special cases or new gags.

# Opt. #7: Represent as spatial partition

- If every criterion is a numeric range x ∈ [a,b], then each one is an axis
- and a rule *R* with *c* criteria is a *c*-dimensional region.
- A query *Q* with *q* facts is a *q*-dimensional point.
- All *possible* fact and criteria keys form a *n*-dimensional space.
- Consider a query *Q* continuously infinite along all *n* axes not in *q*
- Consider each dimension in *n* not present as an axis *c* ∈ R to be an implicit criterion $x \in [ -\infty, \infty ]$
- So finding all rules *R* that match a query *Q* means enumerating which subspaces contain *Q*.

By representing each context as an axis in an n-dimensional space, each possible rule's criterion vector as a c-dimensional subspace, and the query as a q-dimensional point, the problem of selecting responses becomes a spatial interval search for the most specific subspace containing (q), allowing lookups to occur in logarithmic time.

> Ie, consider the R-tree, which is a fast spatially sorted data structure used for *eg* Google Maps queries like "find all restaurants within 2km of here"
> You can use an R* or an X-tree to extend this concept from two to N dimensions.
> The rules database can be built offline, so insertion performance and unbalanced trees aren't a problem: you do the additional work to precompute perfectly balanced trees before the game ever runs.

Maybe I invented this just so I could say "Q continuum"  at work

134

# Left4Dead2 contextual player dialog



Players could actively trigger dialog by selecting it from this wheel. This would correspond to the "concept" of the speech query, and then the other contexts pick the specific line.

# Dynamic Scripted Interactions

- In addition to animation and speech, you can also trigger entire scripts.

- Query facts can be bound to script keys

- Eg, for a "push button" script that has parameters

```
        < $anim_name ; $speaker ; $target_object >



                     Query: {concept=SEE_OBJ, who=redbot, target=button }
-> match { "button_script" }
```

136

# Accidentally Turing-complete

You can do *any* kind of logic
with this system; it has

- An alphabet of symbols
- A state tuple
- Rules for
  - Writing to the state tuple
  - Conditional branches

*(ok, technically it is a Minsky register machine
but that is Turing-equivalent)*
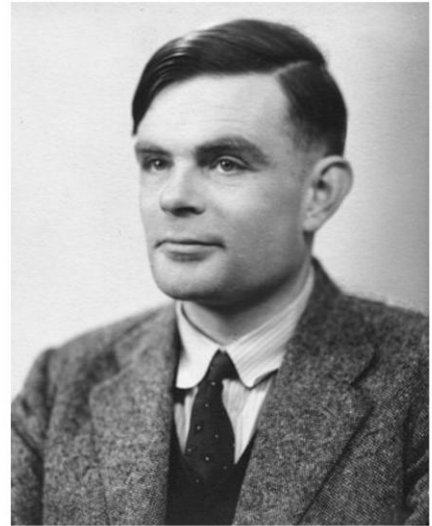


Incidentally this is enough to build *any* general-purpose program. It's turing complete.
With conditional branches and stored state, you can do actual logic and computation
in the system.
That makes it possible for writers to implement flow charts in the dialog engine...
which is the essence of a conversation tree
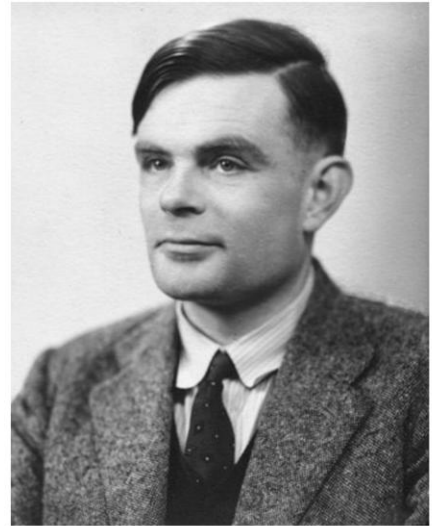        Or running gags, or followup comments, etc.


Picture of Alan Turing via Wikipedia

137

# Accidentally Turing-complete

A domain-specific language

- Rules-based
- Comfortable for dialog
- *And* fully expressive scripting!

- Tastes great
- *And* less filling!



Thinking in rules is writer friendly but the fact that system is turing-complete means that you can express any logic with it. Ie, the system is a fully expressive domain-specific language.

(too cumbersome for general-purpose code, but can be stretched to accommodate any special case)

Picture of Alan Turing via Wikipedia

# in Pseudo-Script…

```
g_ResponseDB.AddRule(
   criteria = [ Criterion( "concept", λ(concept) { concept == "isClimbing" } ) ],
   response = λ(speaker) {
    speaker.playRandomWav( [ "vo.climbing.1", "vo.climbing.2", "vo.climbing.3" ] )
   }
)

// and a more specific rule if it's snowing
g_ResponseDB.AddRule(
   criteria = [ Criterion( "concept", λ(concept) { concept == "isClimbing" } ),
                Criterion( "isSnowing", λ(fact) { fact >= 1 } ) ],
   response = λ(speaker) {
    speaker.playRandomWav( [ "vo.climbing_snow.1", "vo.climbing_snow.2" ] )
   }
)
```

Example bindings to Squirrel, a scripting language we're experimenting with

# Back in History: Half Life 2's "response rules" system

- A small set of general speech concepts
  - "reloading", "help me", "covering"
- Specialized by certain character criteria
  - Gender of character, current map
  - Optional factors: presence of enemies, health of player

Back in history : the Half Life 2 "response rules system"
>    A very simple database of general speech concepts like "reloading", "help me", specialized by a small set of criteria: gender of speaker, current map, some optional factors like the presence of enemies or health of player.

# Back in History: Half Life 2's "response rules" system

*vortigaunt.cpp*

```cpp
void CNPC_Vortigaunt::Use( CBaseEntity *pActivator,
  CBaseEntity *pCaller )
{
  // If we haven't said hi, say that first
  if ( !SpokeConcept( TLK_HELLO ) )
  {
    Speak( TLK_HELLO );
  } else {
    Speak( TLK_IDLE );
  }
}
```

Back in history : the Half Life 2 "response rules system"
  A very simple database of general speech concepts like "reloading", "help me", specialized by a small set of criteria: gender of speaker, current map, some optional factors like the presence of enemies or health of player.

So you had just a single "TLK_HEALING" event, and then the system would try to pick the best, most specific line automatically based on all the other factors

# Back in History: Half Life 2's "response rules" system

*scripts/response/vortigaunt.txt*

```
response "VortigauntIdle"
{
   scene "scenes/npc/vortigaunt/poet.vcd"
   scene "scenes/npc/vortigaunt/hopeless.vcd"
   scene "scenes/npc/vortigaunt/alldear.vcd"
   scene "scenes/npc/vortigaunt/prevail.vcd"
   scene "scenes/npc/vortigaunt/seenworse.vcd"
   scene "scenes/npc/vortigaunt/persevere.vcd"
   scene "scenes/npc/vortigaunt/worthless.vcd"
   scene "scenes/npc/vortigaunt/whereto.vcd"
}

rule VortigauntTlkIdle
{
         criteria                IsVortigaunt ConceptTalkIdle NotInCombat
         response                VortigauntIdle
}
```

Back in history : the Half Life 2 "response rules system"
    A very simple database of general speech concepts like "reloading", "help me", specialized by a small set of criteria: gender of speaker, current map, some optional factors like the presence of enemies or health of player.

So you had just a single "TLK_HEALING" event, and then the system would try to pick the best, most specific line automatically based on all the other factors

# Feature I wish I had thought of

- Self-disabling rules
  - "Don't say this more than once"
  - "Don't say this if it's been said in the last 60 seconds"
- Had this on *responses*, but that meant a rule would match but no voice would play.
- Writers had to make a special-case variable for each one-off line.

Storing individual variables to remember which lines were said, and having to add criteria on the rules to prevent them matching twice, is lame. It would have been better to have some way for a response to remove itself from the database after playing.

# Why end-of-line callbacks are important

- Source had a design flaw where speech couldn't call back to gameplay when it was done
- Hard-coded delays for followup lines
- Required localized voice actors to match timings exactly
- Speech/anim system must call back into the response database when the line is finished
  - Relying on timers and dead reckoning is clumsy