Game Architecture

Presented by: Marcin Chady

RADICAL ENTERTAINMENT
a Radical Entertainment Presentation.

Introduction

- When you sit down and start typing code, what do you type first?
  - And what language do you type it in?
- Need to establish basics
  - Input comes in
  - Time passes
  - Rendering and sound go out
- Elements
  - Major subsystems
  - Main loop
  - Time
  - Game state
  - Communication
  - Entities

Language

- 15 years ago there was only one answer to the question "What language do I write my game in"
  - C / Assembly
  - Maybe C++, without advanced features, if you were feeling really bold
- But you can get lots of benefits from using a higher level language
  - Expressiveness, iteration, safety
  - More accessible to designers (and end users)
  - It all comes down to development speed

Language Performance

- Performance is always a concern
  - Use the highest level language that has “good-enough” performance
  - Remember, even C/C++ are a compromise
    - You can always do it a little faster in assembly
    - as long as you don’t mind your application taking an order of magnitude longer to write
- Games usually try to play it a little safe and use a mixture
  - C/C++ “engine”
  - a high level language bolted on for “scripting”
    - We’ll talk about scripting in another lecture
- Not all though
  - “Jak & Daxter” is written almost entirely in a Lisp-derivative

The Game Loop

- What does a game do?
  - Takes user, and maybe network inputs and generates a displayed frame, and some sound effects
  - This involves the co-ordination of dozens of major subsystems, hundreds of minor subsystems, and thousands of entities
- "Outer loop" of a game, handles:
  - Setup/shutdown of the application
  - Managing the high-level game state
    - Front-end, in-game, paused, etc.
  - Controlling overall game flow
  - Updating the major subsystems

Styles

- Like operating systems, the “game kernel” can be structured in several ways:
  - Single-threaded, monolithic
  - Multi-threaded, co-operative
  - Multi-threaded, pre-emptive
  - Client-server

Monolithic

```
while (1)
{
    input();
    simulate();
    render();
    sound();
}
```

Co-operative tasks

```
class Task
{
    virtual void Run() = 0;
};

class Renderer : public Task
{
    void Run(float time);
};

class TaskManager
{
    void RunTasks();
    void AddTask(Task*);
};

void TaskManager::RunTasks()
{
    foreach(task)
        task->Run();
}
```

Pre-emptive

```
void InputThread()
{
    while(1) input();
}

void SimulationThread()
{
    while(1) simulate();
}

void RenderThread()
{
    while(1) render();
}

void SoundThread()
{
    while(1) sound();
}
```

Issues

- Must handle multiple high level game states
  - Front-end, in-game, paused, etc.
- Must ensure correct order-of-operations
  - E.g. perform collision detection before AI state update
  - Explicit / fixed ordering
  - Priority
  - FIFO
- Must ensure consistency of game state across sub-systems
- Should maintain modularity, minimize dependencies, reduce bugs, etc.
- Each system has benefits and drawbacks

Comparison

- Monolithic is easy to code up, but can get messy
  - Flow-of-control can be complex
  - Top-level may have too much knowledge of underlying systems
  - Maintenance problem
- Cooperative tasking systems are flexible, but clarity suffers
  - What happens in what order difficult to discern by examining code
  - Can be too much flexibility
- Pre-emptive systems are tough to get right
  - Complex interprocess communication
  - Deadlocks, race conditions
  - Questionable performance if used extensively
  - Increasingly parallel hardware makes this a major area for the future

In Practice

- Games tend to use multiple approaches:
  - Systems that manage asynchronous resources run in separate threads
    - Low-level input
    - Sound
    - File I/O services
  - Entity systems often use cooperative tasking, but separate from the main loop (inside the AI)
    - Script languages sometimes have this built in
  - Other systems may use cooperative multitasking or are explicitly coded
- One size doesn't fit all
  - Don't re-write Linux to make your game work

Time

- One of the most important aspects of game architecture is time
- How time is handled should be designed into the architecture from day one
- Inconsistent handling of time throughout the game will create difficult to fix bugs, and frustrating game-play
- Be clear as to the units of time used in variable names and function calls

```
void Update( float time ) { ... }  // Bad
void Update( float milliseconds ) { ... } // Still bad
void UpdateMilliseconds( float time ) { ... }  // Good
void Update( Time& time ) { ... }  // Good
```

Wall-clock vs. Game-clock

- Rule of time: game time is constant regardless of frame rate
  - "Game time" must always be aligned to "real time"
    - A car traveling at 100 km/h must travel the same distance when the game is updating at 60 fps as it is when the game is updating at 20 fps
    - Not 33 km/h at 20 fps!
  - In some situations "game time" may run at a different speed
    - In sports games, clock is often accelerated
    - When game is paused, "game time" doesn't change
- Old PC games often got this wrong
- Game architectures can (and do) achieve this in different ways

Subtleties of Time

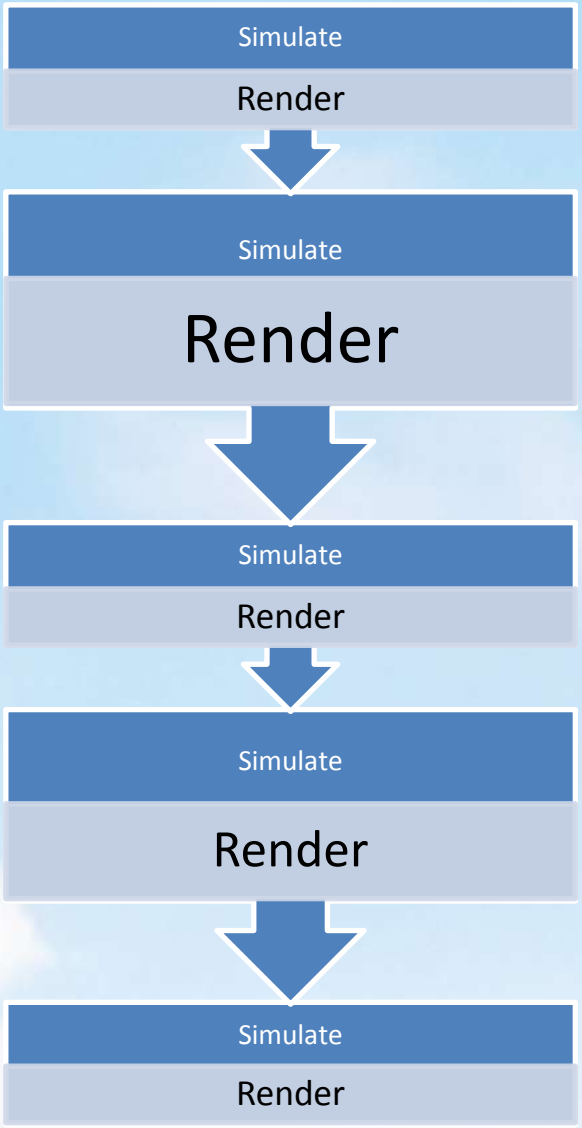
- We typically discern several different clocks
  - Real
  - Audio
  - Simulation
  - Game
- Various game states affect clocks in different ways
  - E.g. pausing pauses the simulation clock only
- Many games cap time advancement in some way
  - Don't want to advance if stopped in the debugger
  - If frame rate gets really bad, slow-mo may be acceptable
  - Some subsystems may not tolerate large time deltas
- Need to be cautious when interacting systems use different time scales
  - If a cinematic needs to by synced to sound (e.g. for lip-sync), may need to drive it directly by the audio clock

Time – Fixed step

- The simplest approach is the fixed time step:

```
while (1)
{
    time += timestep;
    simulate(timestep);
    render();
}
```

Time – Fixed step



- Advantages:
  - Simple
  - Repeatable behaviour
  - Great for debugging, replay
- Disadvantages
  - Sensitive to variable work loads
    - Really needs fixed frame rate!
- One size doesn't fit all
  - Physics needs small deltas
  - AI can deal with larger ones
  - But small steps use up a lot of CPU

Time – Multiple simulation steps

- The fixed time step can be used with multiple simulation steps to deal with slow frame rates:

```
while (1)
{
    now = systemTime();
    while (simtime < now)
    {
        simtime += timestep;
        simulate(timestep);
    }
    render();
}
```

Time – Multiple simulation steps

- Advantages
  - Still fairly simple
  - Deals well with poor frame rate in some situations
    - e.g. when the rendering is slow and simulation fast
- Disadvantages
  - Everything runs in a multiple of the time step
    - Must be careful to correctly propagate errors across updates
    - Rendering rate locked to time step
  - Unstable
    - If the simulation step is slow, running it multiple times will create more work, which will make the delta greater, which will crate more work, and so on, leading to…
    - "Death spiral!"
    - Fix "death spiral" by putting a cap on update rate
      - Of course, this will result in visual stutter

Time – Variable step

- Another possibility is the variable time step:

```
while(1)
{
    now = SystemTime();
    timestep = now - time;
    time = now;
    simulate(timestep);
    render();
}
```

Time – Variable step

- Advantages
  - Self-balancing
  - Works well on platforms where the CPU power is variable (i.e. PC)
- Disadvantages
  - Can lead to instability in code that relies on a small time step
    - Collision detection, response
  - Non-deterministic
    - Can result in unrepeatable behaviour
    - Not as good for instant-replay

Sub-iteration

- A sub-system may “cut up” the variable or fixed step into smaller fixed steps.

```
collision_detect(timestep)
{
    while (timestep > 0)
    {
        dostuff(smallerstep);
        timestep -= smallerstep;
    }
}
```

Sub-iteration

- Advantages
  - Flexible
    - Systems can operate using a fixed or variable step as needed
  - Sensitive systems can use a smaller step
- Disadvantages
  - Increased complexity
  - Systems operating at different update rates may experience undesired interactions
    - Temporal aliasing
    - Shows as a jerkiness, or strobing effect


RADICAL
ENTERTAINMENT
PAUSED

Game Architecture
Part 2

Overview

- Game state representation
- Entities
- Communication

Major subsystems

- Video games have a fairly natural decomposition into subcomponents
- Major systems should have well defined boundaries and communication paths
  - They are probably going to be implemented by different people
  - Need to manage the complexity a little

List of Major Subsystems

- Input
- Networking
- Rendering
- Sound
- Script
- Loading
- Front-end
- HUD
- Physics
- AI/Gameplay

Modularity

- Ideally, we want the major subsystems to be as modular (decoupled) as possible
  - Each system should view other systems as a black box, regardless of how complicated they are internally
  - Communication should occur through narrow, well-defined interfaces
- All the systems are collaborating on updating or presenting the evolving state of the game
- Designing this game state while maintaining modularity is the fundamental architectural challenge when designing a game engine

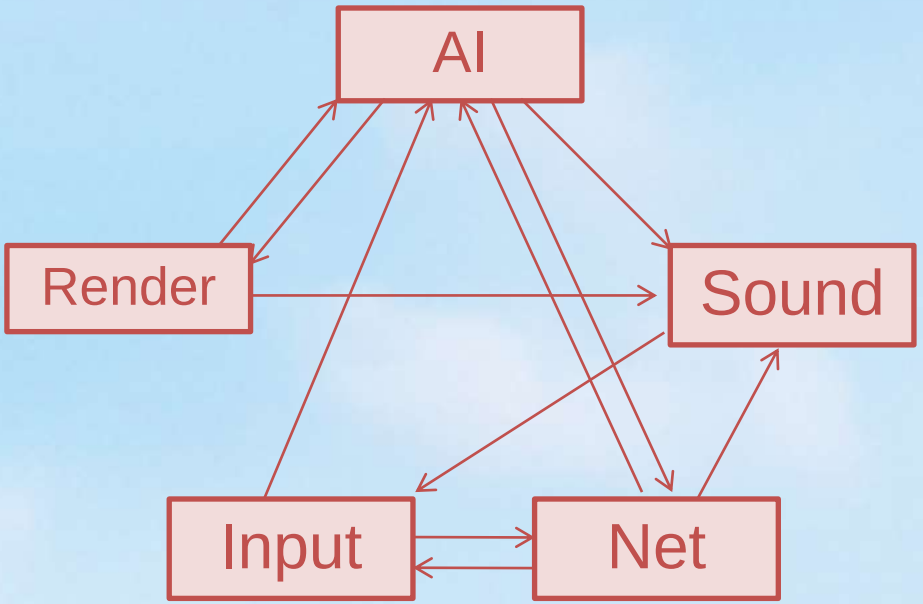
Game State

- The collection of information that represents a snapshot of the game universe at a particular time
  - Position, orientation, velocity of all dynamic entities
  - Behaviour and intentions of AI controlled characters
  - Dynamic, and static attributes of all gameplay entities
  - Scores, health, powerups, damage levels, etc.
- All sub-systems in the game are interested in some aspect of the game state.
  - Renderer, Physics, Networking, and Sound systems need to know positions of objects
  - Many systems need to know when a new entity comes into or goes out of existence
  - AI system knows when player is about to be attacked
    - Sound system should play ominous music when this happens
    - Note this is different from playing a sound on impact (event)

Sharing State

- How is the game state made available to sub-systems?
  - Global state
  - Push/Pull (client server) based models
  - Brokers
  - Broadcast-listener
  - Database
  - Shared entities
- Can also use different systems for different parts of state

Global State



- Everyone has access to everyone else
  - If the renderer needs access to player location, it just grabs it from the AI sub-system
- This is what will emerge if thought isn't given to the design in advance
- A real game has many dozens to hundreds of boxes

Push-Pull (Client Server)

- Flow of state information is restricted to certain directions. Systems have incomplete knowledge of each other
- Example: Renderer (Push)
  - AI tells renderer where objects are
  - AI tells renderer the damage state of objects
    - Renderer matches damage state to a visual representation
  - Game loop tells renderer how much time has elapsed
    - Renderer updates internal effects, animation list
  - AI tells renderer where the camera is
    - Renderer sets up appropriate graphics hardware state
  - AI tells renderer what the player score is
    - Renderer displays it on the screen
  - The rendering system has no knowledge of the AI

Brokers

- Brokers control the communication between complex sub-systems
- Used to further isolate major systems from each other
- Example
  - A broker is responsible for collecting all rendering-related events from the various game sub-systems and sending them to the rendering sub-system (push model)
  - Alternatively, the rendering system retrieves the information from the broker (pull model)
- A broker may have knowledge of many systems, but it should be itself quite simple, and thus easy to modify
  - The broker serves as an intermediate between systems
  - It shouldn't be responsible for doing a lot of complex work

Broadcast / Listener

- When a state changed occurs that might be interesting to other systems, the system responsible for that state broadcasts it through a central event system
- Systems that are interested register as a listener with the event system
- No direct communication occurs between sub-systems
- Useful for communication across thread / network boundaries
- Expensive
  - Requires storage for messages and listeners
  - Message dispatch requires CPU time and memory access
- Convenient
  - Easily abused

Database

- Necessary when game state does not fit in memory
  - MMOs
- Get some stuff for "free"
  - Persistence
  - Queries
- Nice separation between data and code
- Performance can be a problem though
  - You probably still need to cache live stuff in more conventional objects

Shared Entities

- Individual actors in the world are referenced by multiple systems
- Example
  - AI and Renderer both have a reference to “Character”
  - Renderer view and AI view of character need not be the same class or interface though
- A lot like global state on the surface, but can be much cleaner with proper use of interfaces and inheritance
- Probably the most common way of storing game state
- But don’t use bare pointers!
  - Use handles or smart pointers at least

Entity System

- Many different ways to structure the entity system
  - Single-rooted inheritance hierarchy
  - Multiple inheritance
  - Ownership based
  - Component based
- A modern game must manage many thousands of independent entities
  - E.g. a unit in an RTS game:
    - Position
    - Orientation
    - Health
    - Current orders
    - Animation frame

Single-Rooted Hierarchy Example

```
class Entity
{
    virtual void Display(void) = 0;
    virtual void Think(float time) = 0;
};

class Character : public Entity
{
    void Display(void);
    void Think(float time);
};
```

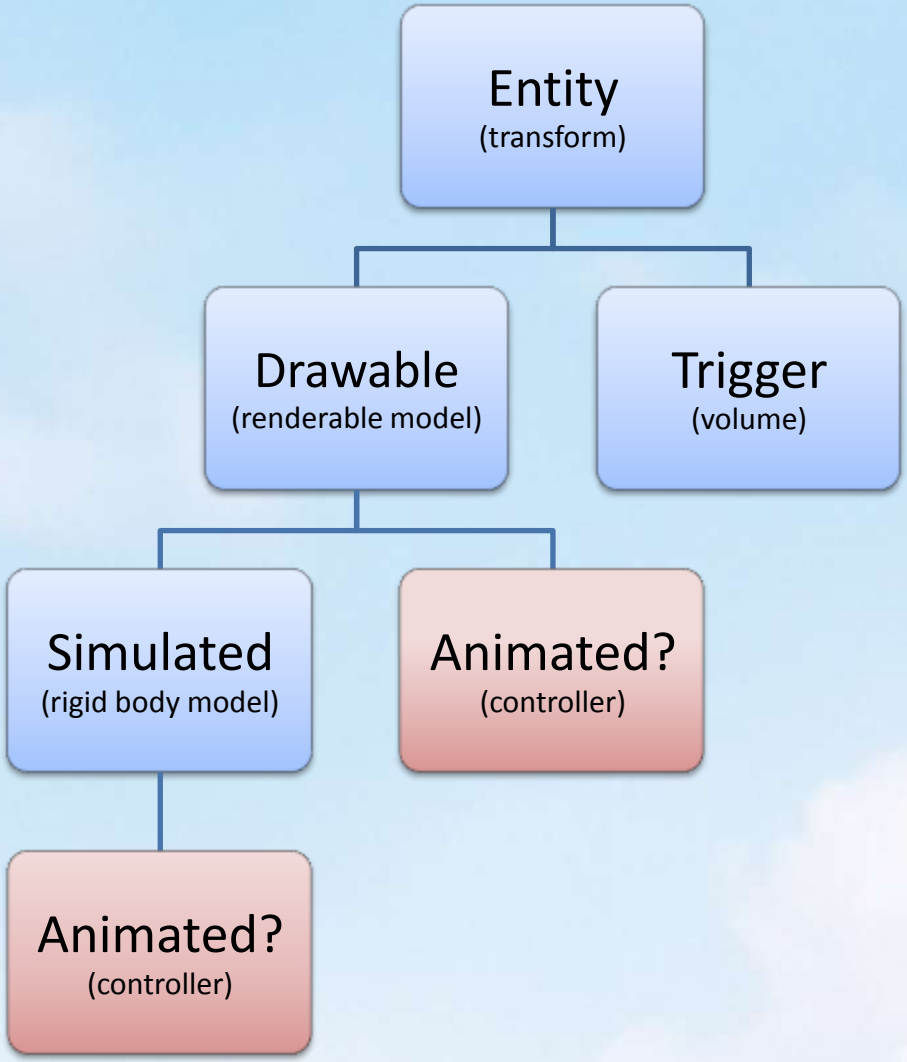
Single-Rooted Hierarchy

- Some base class of entities with virtual functions for common interfaces
  - Update(time), GetPosition(), etc.
- All individual entities derived from this class
  - May be multiple levels:
    - Entity
      - Character
        - » PC
        - » NPC
- Subsystems keep list of entities they care about
  - But only need to know about base classes and/or specializations that are specific to that sub system
- Probably most intuitive way of handling entities
  - Seems very simple at first, but….

Problems with Single-Rooted Hierarchy

- Doesn't scale
  - When relationships get complex things start to break down
    - Hierarchy stops making sense
    - More functionality gets pushed to base class to keep things working
    - May need to start multiply inheriting midstream
  - Hierarchies are difficult to refactor
- Only one base class
  - So requirements of one system tend to take control of hierarchy
    - Usually it's rendering
  - This leads to unneeded functionality
    - AI triggers with display functions
  - Also, divisions that are natural in one system may not be in another

Hierarchy Trouble



Multiple-Inheritance Example

```
class Drawable
{
    virtual void Display(void) = 0;
};

class AI
{
    virtual void Think(float time) = 0;
};

class Character : public Drawable, public AI
{
    void Display(void);
    void Think(float time);
};
```

Multiple-Inheritance Entities

- Base classes for each subsystem, entities multiply inherit
- Scales better than single inheritance
  - Less base class pollution
  - Less conflicting requirements
- Still run in to problems when the hierarchy start getting large
  - Same problems as with single-rooted hierarchy can develop on a smaller scale
  - If hierarchy does have to change, it's painful
- Implementation of multiple inheritance in not straightforward
  - C++ supports it, but it's not for the faint-hearted
    - Behind the scenes stuff is ugly and can cost performance
    - We'll talk about it in the C++ Internals lecture
    - Need to jump though hoops to get things working
      - Virtual inheritance
  - Other languages stay clear of MI or provide limited support

Ownership-Based Example

```
class CharacterDrawable : public Drawable
{
    void Display(void);
};

class CharacterMind : public AI
{
    void Think(float time);
};

class Character
{
    CharacterMind* mind;
    CharacterDrawable* body;
};
```

Ownership-Based Entities

- Entities have no (or very little) hierarchy and farm operations out to other objects
  - "Character" will have pointers to "Renderable", "Simulated", etc.
  - These sub-objects are either obtained from, or registered with the controlling system
- Hierarchy is less fragile than single inheritance or multiple inheritance and less complex
- Unrelated functionality is separated
- Can run into trouble if you let entities develop hierarchy
  - Shouldn't need to though, entities are really simple now
- Some problems though
  - Need lots of wrapper functions
  - Main entity spends a lot of time copying data between sub-objects

Component-Based Example

```
class EntityComponent
{
    virtual unsigned GetID() = 0;
};

class Renderable : public EntityComponent
{
    unsigned GetID();
    void Update(void);
};

class CharacterMind : public EntityComponent
{
    unsigned GetID();
    void Update(float time);
};

class Entity
{
    void AddComponent(EntityComponent*);
    EntityComponent* GetComponent(unsigned id);
};
```

Component-Based Entities

- Generalization of ownership system
  - Entity is totally generic, just a collection of sub-objects
  - Just like ownership, only without knowledge of actual types of owned objects
  - Components can be fetched from an entity by name and/or managed by the interested subsystem
- Much more flexible than code driven hierarchies
  - Can add / remove components on the fly
  - Can build entities from components using data/scripts
- Problems
  - Complicated (and can be slow)
  - Can be inconvenient
    - Hard to communicate with and between components

Component-Based Subtleties

- Need to have sophisticated system for:
  - Updating components
  - Communicating with/between components
  - Sharing data between components
- Otherwise it'll be slow
- Don't derive components from anything other than base component
  - Otherwise you're just creating the fragile hierarchy we are trying to get away from again
- Can implement components in script

Entities Wrap–Up

- Lots of ways to do it
- Single inheritance probably the worst
  - Is only good if you know the hierarchy isn't going to change
    - On a big project you never do
  - It's very quick to put together
  - Still gets used a lot
- Component based system is "best"
  - Most flexible
  - Least fragile
  - Much more complex than other systems though
- MI and ownership are in the middle
  - Both are cleaner than single inheritance
  - Require more discipline and are less flexible than components
- Any system requires initial design and dedication to keeping the hierarchy clean
- More on this in Marcin's talk from GDC Canada 2009

Communication Strategies

- Subsystems and entities need a way to talk to each other
- Several ways to do this
  - Direct function calls
  - Shared virtual interfaces
  - Events
  - Callbacks

Function Calls

- Pros
  - Simple
  - High performance
- Cons
  - Scales poorly
  - Header file dependencies increase
    - Greater build times
  - High coupling
    - Interfaces between objects widen over time
- Brokers can reign in complexity and keep the interfaces narrow and focused

Virtual Interfaces

- Pros
  - Reduces coupling
    - Systems only joined at the shared virtual interface
  - Gives you polymorphism if you need it
- Cons
  - Somewhat more complicated
    - Separate interface from implementation
  - Interfaces that are too narrow are painful to use in practice
  - Extra layer of indirection
    - Slightly slower to call

Events

- Complicated to implement/use
  - Really good systems are type-safe, and double-blind
    - Example: Qt signals and slots
    - Quite involved to implement
  - Poor systems (like WndProc) are usually untyped
    - Lots of conventions and casts
- Low coupling / very good scaling
- Need to be careful about “dangling events”
  - If a system is waiting for an event that never fires, it can be very difficult to debug

Callbacks

- One way to allow objects to communicate is to pass function addresses to each other
- Many different implementations
  - Function pointers
  - Callback objects/functors
- Pros
  - Offers the decoupling of an event from the communication of it
  - Functors look and act like function calls with return values
  - Stronger contract than pure events
- Cons
  - More work to hook up between objects
  - Not a complete system on their own
    - Needs another type of interface to set up

Other Things to Consider

- Multicasting
  - May want to send a message to multiple things at once
    - Example : If player makes noise, all enemies AI in “hearing radius” get an event
- When to dispatch
  - May want to defer events, and send only during particular points in the game loop
  - May want to cache and defer for multi-threading
- Adaptation
  - In many cases, systems want to morph events to some degree
    - Example : Game AI says “Character is falling”, sound system turns that into “play scream07.wav”.

Conclusion

- To help manage the complexity, it's important to divide the games into simpler components
  - Minimizing coupling is a good thing
    - Both at the system and entity level
    - Formalized communication through designed interfaces is superior to rooting around in someone's private data
- Make the AI authoritative
  - AI systems push (via brokers), broadcast or have components owned by entities
- How you go about doing this will have a huge impact on the effort involved in building your game.
- Most games never manage architectural purity
  - but if you start strong the problems in the end will be mostly cosmetic

RADICAL
ENTERTAINMENT