Game Architecture

Introduction

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

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

Game Loop Styles

- Several common ways to structure game loop / update of high level components
  - Fixed monolithic
  - Flexible tasks
  - Multi-threaded

Monolithic

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

Questions
- What happens when the game is paused?
- How do I maximise the use of multiple CPU cores?

Flexible Tasks

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

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

vector<Task> tasks;
while(1)
{
    for(Task t : tasks) t->Run(time)
}
```

Multi-threaded

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


void main()
{
    std::thread t1(InputThread);
    std::thread t2(SimulationThread);
}
```

Questions
- How do I pass data from one thread to another?
- How do I ensure that data is passed only when it's ready?

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
- Task 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 inter-process communication
  - Deadlocks, race conditions, memory stomps
  - Questionable performance if used extensively
  - Increasingly parallel hardware makes this the rigueur du jour in high-performance games

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 flexible task systems, but below the level of the main loop (inside the gameplay/AI subsystem)
    - Script languages sometimes have this built in
  - Other systems may use non-preemptive multitasking or are explicitly coded
- One size doesn't fit all
  - 'AAA' games rely heavily on multithreading (modern consoles are all multi-core)

Other Game Loop Considerations

- Some event-driven platforms may not be able to loop
  - Any platform with a GUI suffers from this to some degree
- Networking
  - What if one of the subsystems is running on a different machine
- Fine grained parallelism
  - May need a parallel task queue system to spread work uniformly across many cores

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
  - Space Invaders actually used this "on purpose"
- Game architectures can (and do) achieve this in different ways

Subtleties of Time

- We typically discern several different clocks
  - Real
  - Simulation
  - Playtime
  - Other systems specific (i.e. audio)
- 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, easy to understand
  - 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 variable work loads:

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

Time – Multiple Simulation Steps

- Advantages
  - Still fairly simple and intuitive
  - 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
    - Rendering rate locked to time step (temporal aliasing)
  - Unstable
    - If the simulation step takes longer than `timestep`, the loop will overrun `systemTime()`, which will create yet more work, and so on, leading to...
    - "Death spiral!"
    - You can fix it by putting a cap on simulation iterations, but
      - 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 in games where CPU load is variable (e.g. open-world games)
- Disadvantages
  - Assumes that the time it took to compute the last frame is a good approximation of the next
    - Not a safe assumption by any stretch
  - Can lead to instability in code that relies on a small time step
    - Collision detection and 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

Conclusions, Part 1

- Game loop
  - Think about how high level subsystems are driven
- Time
  - Must have a consistent way of managing time
  - And distributing to all subsystems

Break

Game Architecture
Part 2

Overview

- Subsystems & 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

List of Major Subsystems

- Input
- Networking
- Rendering
- Sound
- Script
- Content Loading
- HUD/Front-end
- Physics
- Animation
- 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 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 building 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)

Private State

- Some state can be kept private to systems
- Always good to make state private where possible
  - Easy to reason about
  - Reduces bugs
- Some pitfalls though, easier to create certain classes of logic errors
  - Divergent states in different systems
  - Bad selection of private state can lead to worse solutions (drilling holes) when state needs to be publicized on short notice.

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
- Often, what we call 'AI' or 'Gameplay' is the authoritative source.

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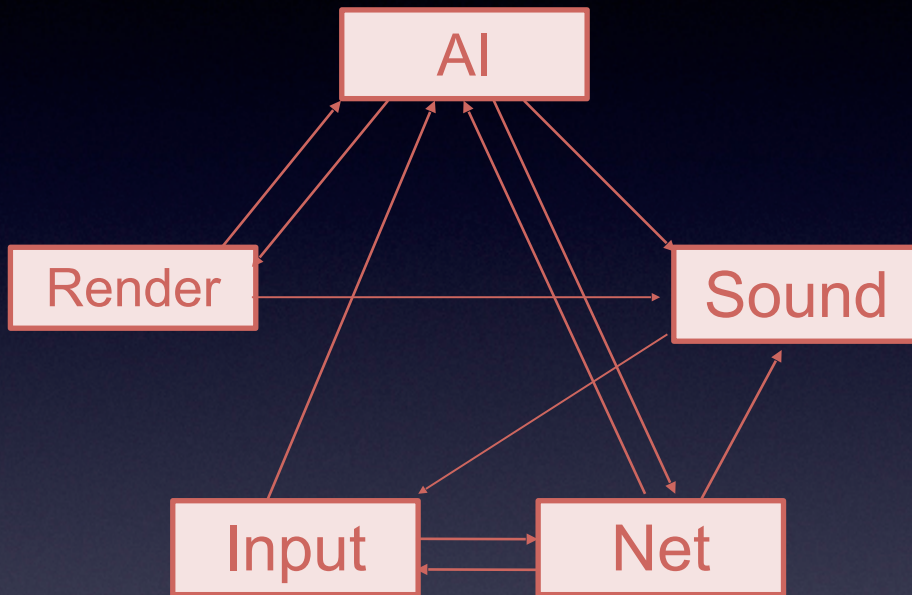
- Everyone has access to everyone else
  - If the renderer needs access to player location, it just grabs it from the AI sub-system
- This is what will emerge if thought isn't given to the design in advance
- A real game has many dozens to hundreds of these boxes

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 matrix state for rendering
  - 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

Broadcaster / 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
- Will dive into even more detail in a minute

Communication Strategies

- Subsystems and entities need a way to talk to each other
- An extension of game state sharing mechanisms
- 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
  - Wide interfaces increase coupling again
  - 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
- Event systems don't have to be asynchronous
  - Can be function calls to a broker that executes a handler immediately
- 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)
  - Lambdas
- Pros
  - Offers the decoupling of an event from the processing of it
  - Functors look and act like function calls with return values
  - Stronger contract than pure events
- Cons
  - Less readable code, even with lambdas
  - 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 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".
  - Beware of snowball effects

Entities

Almost all games use some kind of entity system (even if they also use some of the other strategies for state management)

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

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


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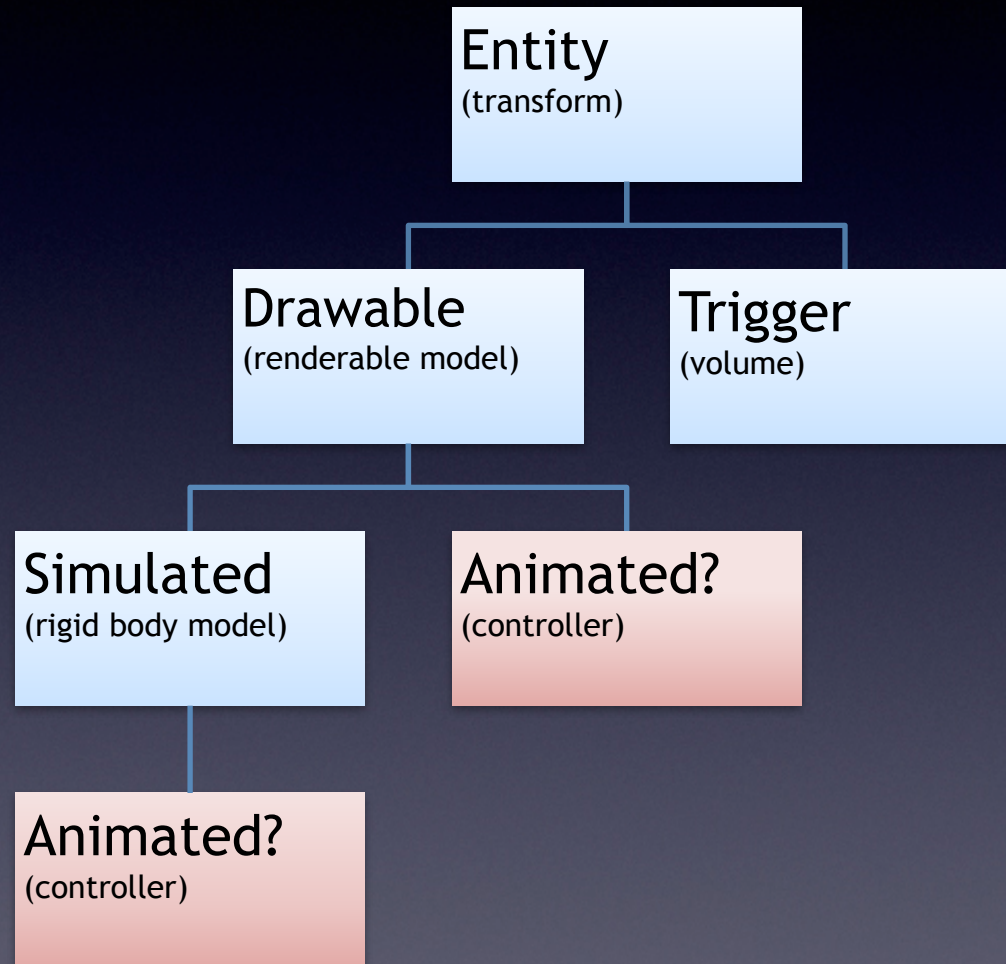
Single-Rooted Hierarchy

- 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
- Subsystems keep lists 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 multiple-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

Entity
(transform)

Drawable
(renderable model)

Trigger
(volume)

Simulated
(rigid body model)

Animated?
(controller)

Animated?
(controller)

Multiple-Inheritance Example

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


class AnimatedDrawable : public Drawable {
    void Display(void);
};


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


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

Multiple-Inheritance Entities

- Base classes for each subsystem, entities multiple-inherit
- Scales better than single inheritance
  - Less base class pollution
  - Less conflicting requirements
- Still problems when the hierarchy start getting large
  - Same problems as with single-rooted hierarchy but 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++ Pitfalls lecture
    - Need to jump through hoops to get things working
      - Virtual inheritance
  - Other languages stay clear of MI or provide limited support

Ownership-Based Example

```cpp
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 to extract relevant info from the entity object
  - Main entity spends a lot of time copying data between sub-objects

Component-Based Example

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

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

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

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

Component-Based Entities

- Generalization of ownership system
  - Entity is completely generic, just a collection of sub-objects
  - Just like ownership, only without knowledge of actual types of owned objects
- 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 Subtleties

- Need to have sophisticated system for:
  - Updating components
  - Communicating with/between components
  - Sharing data between components
- Otherwise it'll be slow
- Can implement components in script
- 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
- Allowing direct fetch of components is seductive, but dangerous in the long term
  - Messaging between components is much better

Component Fetch vs. Events

```
// Bad
DamageComponent* damage = entity->getComponent("damage");
damage->takeDamage(10);

// Good
DamageEvent* damage(10);
entity->sendEvent(damage);
```

ECS (Entity-Component-System)

- The new hotness as far as ways to implement a component-style system
  - Terribly named though, hard to distinguish from previous takes on components.
  - Pioneered (in it's modern form) by Mike Acton at Insomniac
  - Unity, and other big players, moving towards this
- Conceptually same as component systems we've already talked about, but inside out
  - "Entities" : just an id rather than a 'real' object
  - "Components" : just data, stored in efficient-to-iterate lists
  - "System" : Logic lives here, each system iterates and updates a set of related components.
- Systems can message each other, or use temporary components on the entity to communicate

ECS Architecture

Physics

Physics components

Renderer

Render components

Entity

Component pointers

AI

AI components
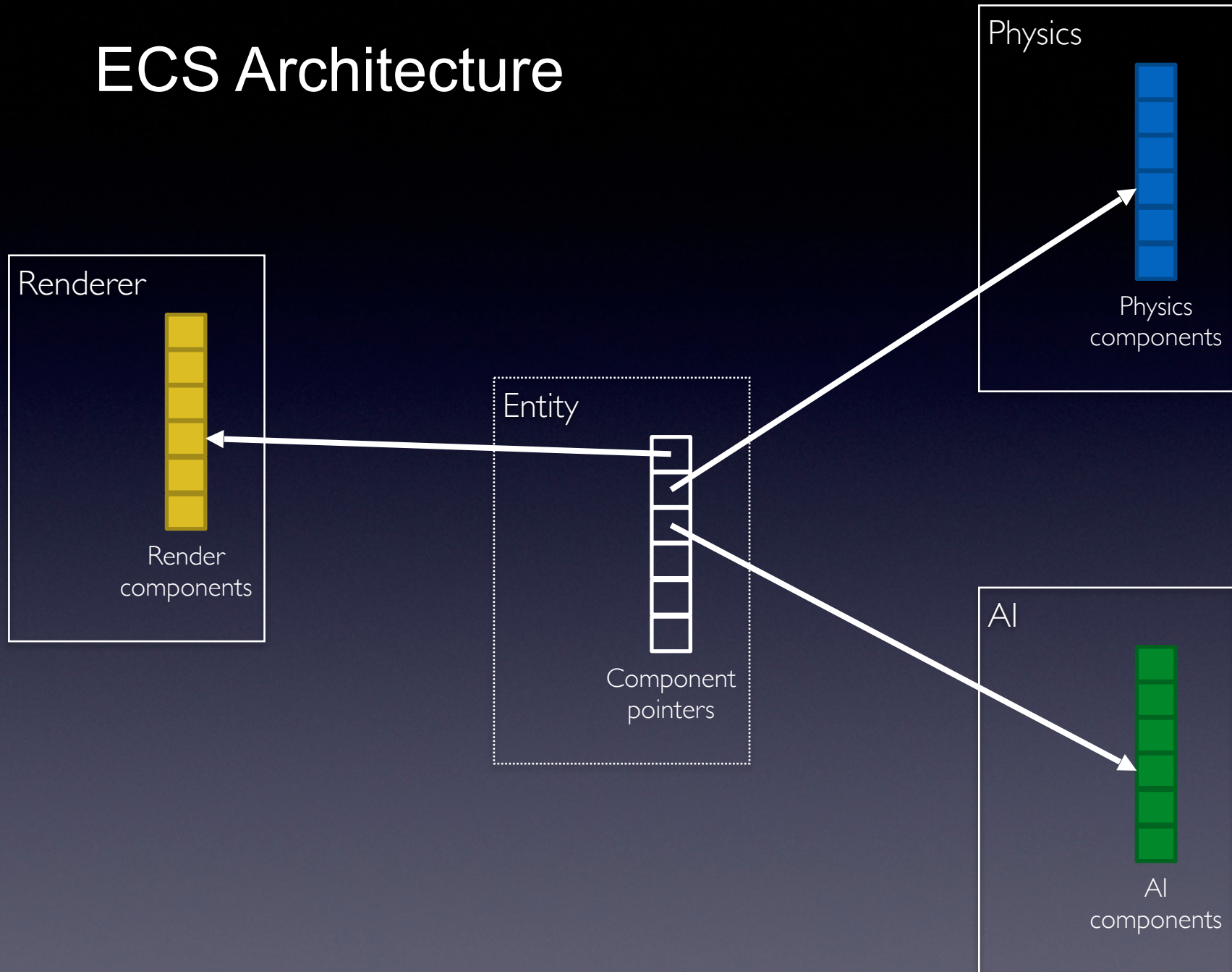
ECS Pros and Cons

- Pros
  - Fast : Cache-friendly, drastically reduces function calls
  - Safe : Less opportunity for dangling pointers, reference cycles
  - Systems can be completely independent and can do as much or as little work as you like – minimal coupling, cleaner code
  - Order of systems is much less important – if you get "target detection" and "target selection" round the wrong way, you just get a frame's delay in things happening
  - Very parallelizable - can calculate dependancies based on what components are read and written
- Cons
  - Makes your head hurt
  - Messaging between systems can be hard

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 (it will)
  - 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
  - ECS-style implementations have lots of advantages, but are conceptually complicated
- Multiple inheritance 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

Conclusion

- To help manage the complexity, it's important to divide the games into simpler components
  - Minimizing coupling is a good thing
- Have decoupled communications systems in place
- Consider how state/entities are stored
- 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