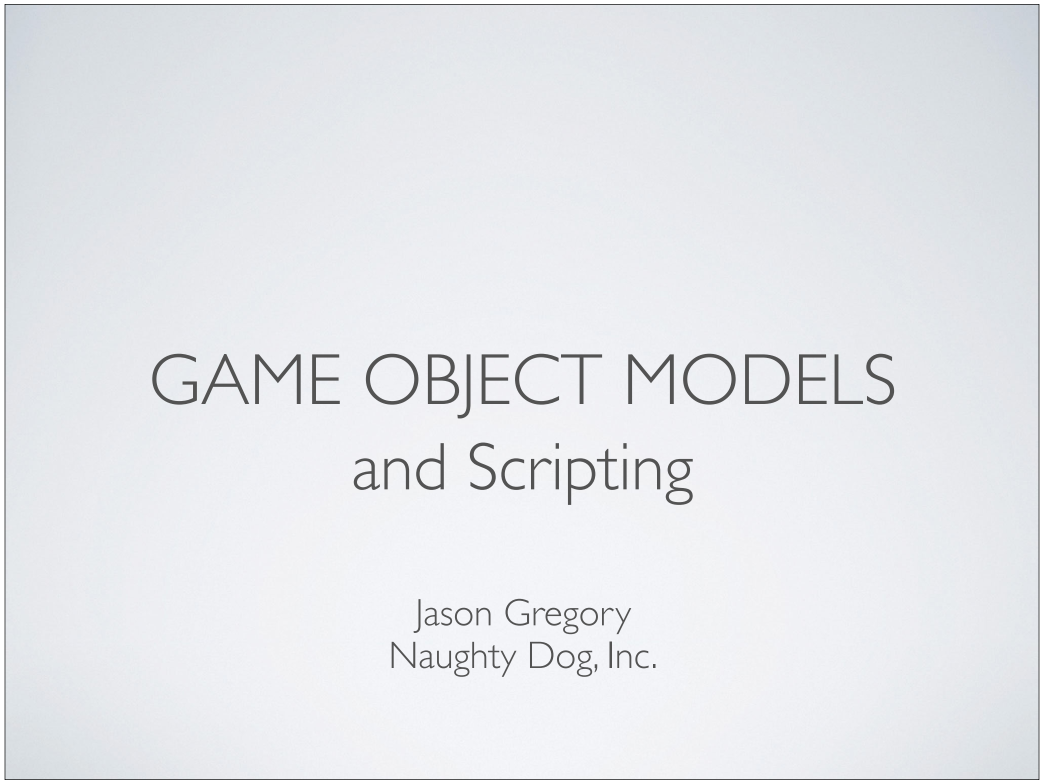
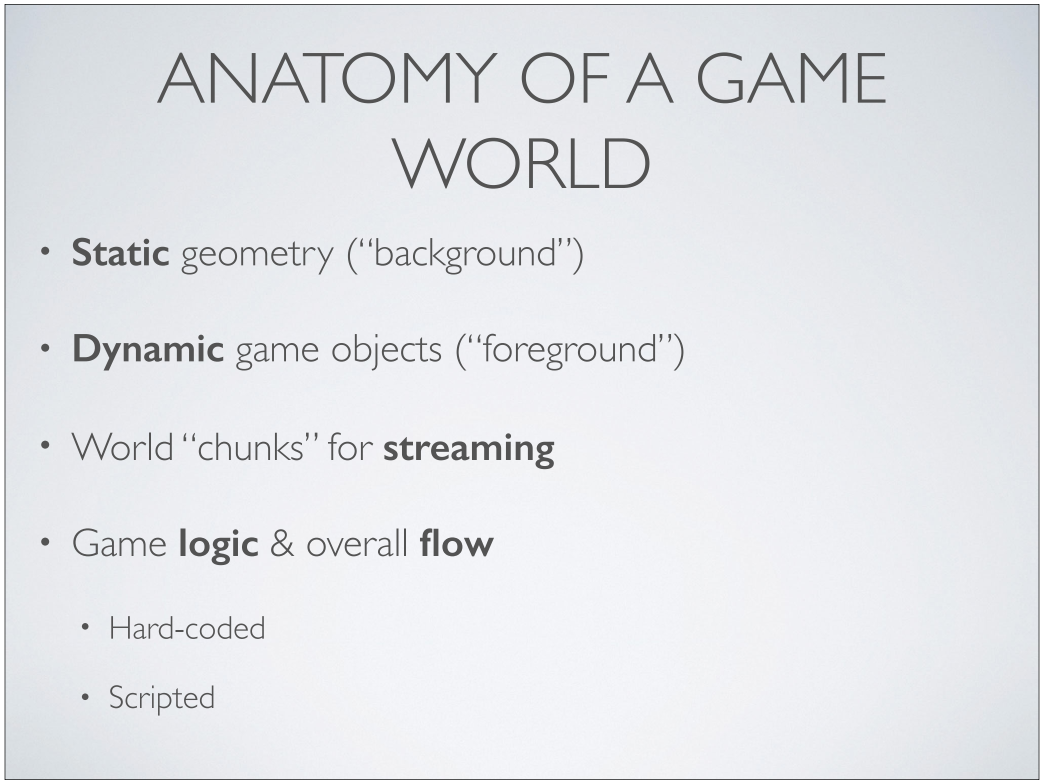


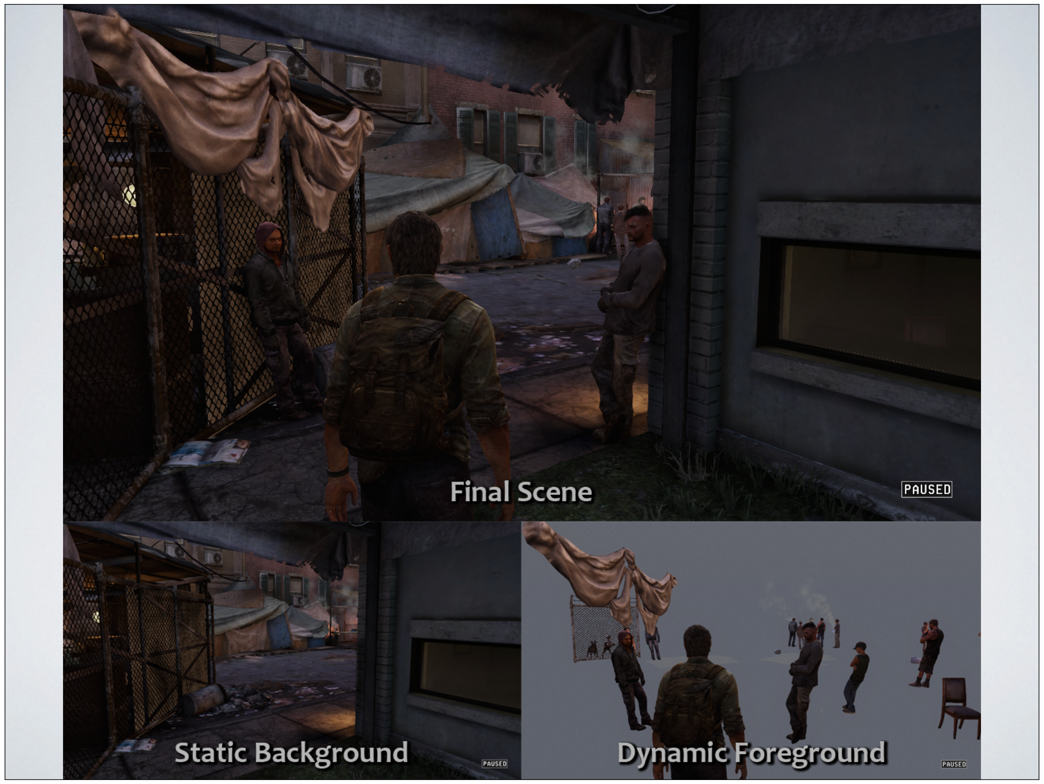
GAME OBJECT MODELS
and Scripting

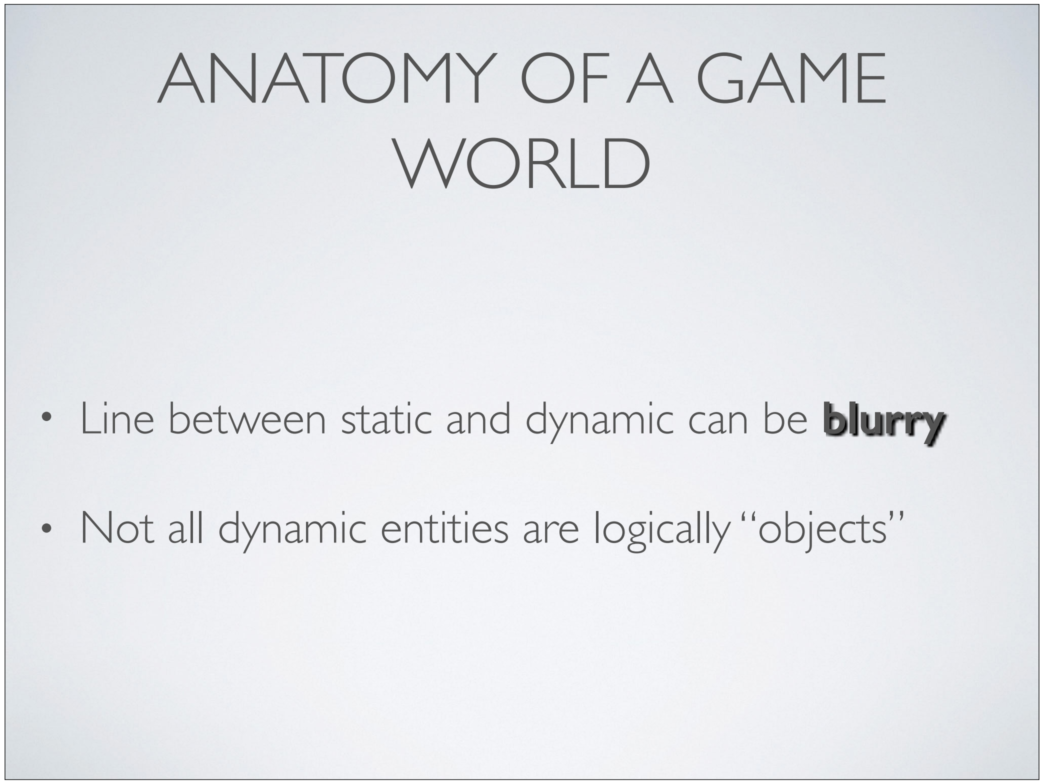
Jason Gregory
Naughty Dog, Inc.



ANATOMY OF A GAME WORLD

- **Static** geometry (“background”)
- **Dynamic** game objects (“foreground”)
- World “chunks” for **streaming**
- Game **logic** & overall **flow**
  - Hard-coded
  - Scripted


Final Scene
PAUSED
Static Background
PAUSED
Dynamic Foreground
PAUSED

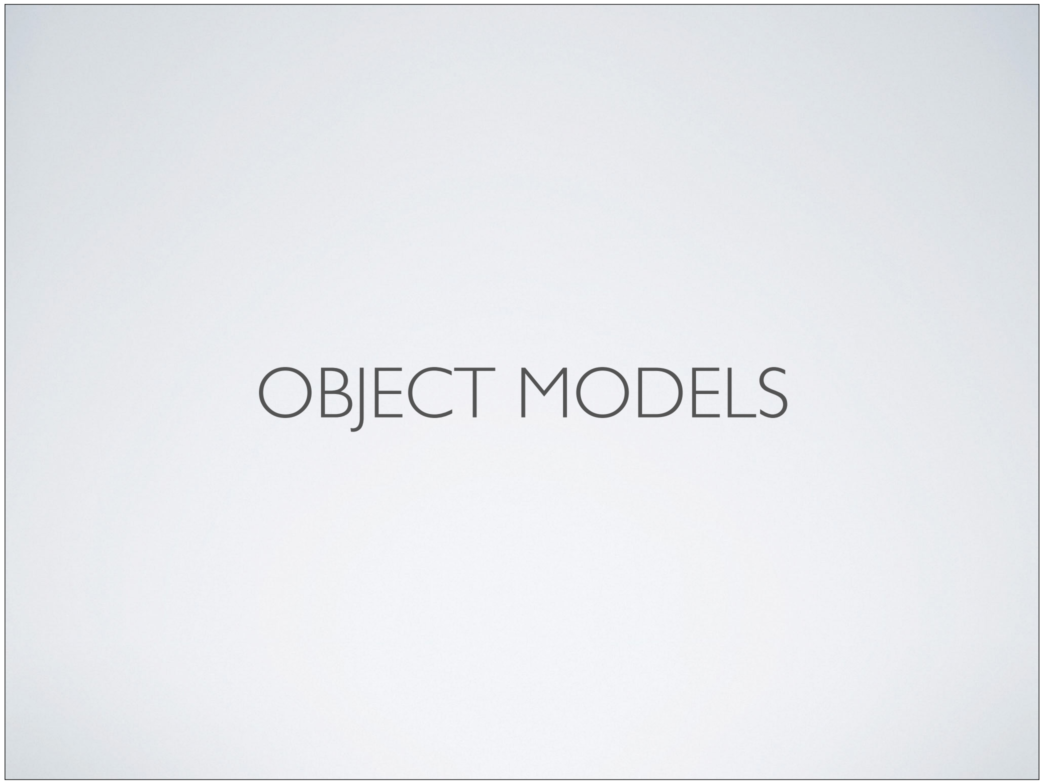


ANATOMY OF A GAME WORLD

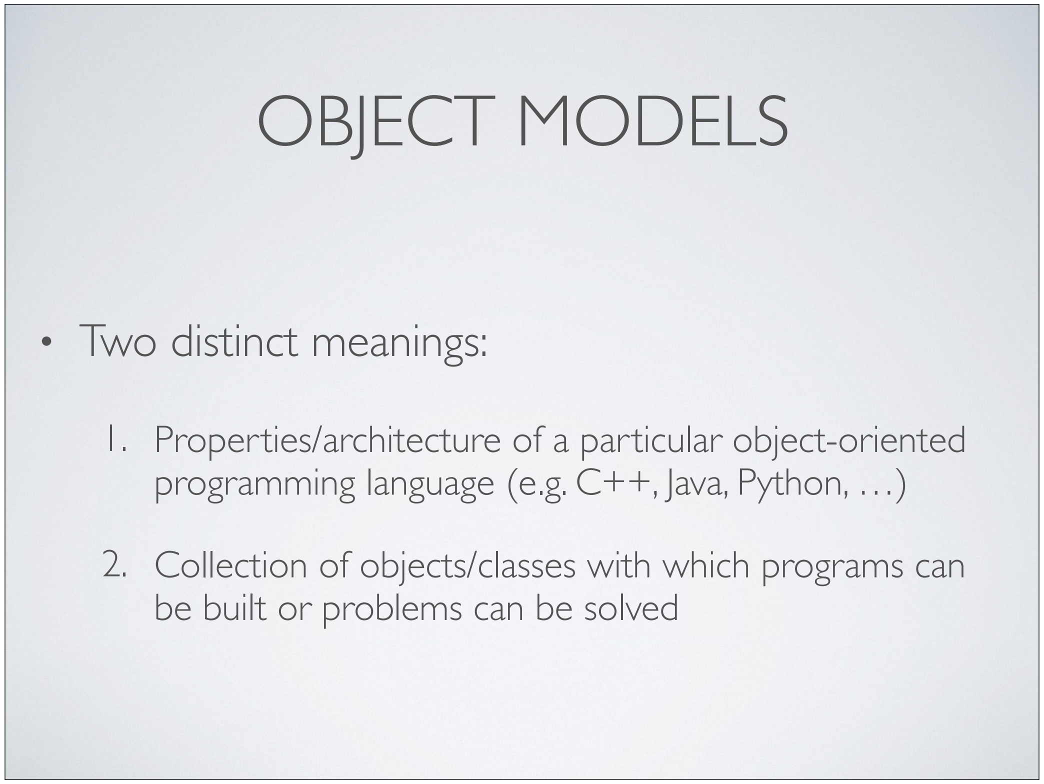
- Line between static and dynamic can be **blurry**
- Not all dynamic entities are logically "objects"

particle effects, water, lights: might be BG or FG

Dynamic lights, running instances of a script, timers, …
May be implemented as a dynamic "object" despite not representing any logical "object" in the game



OBJECT MODELS



OBJECT MODELS

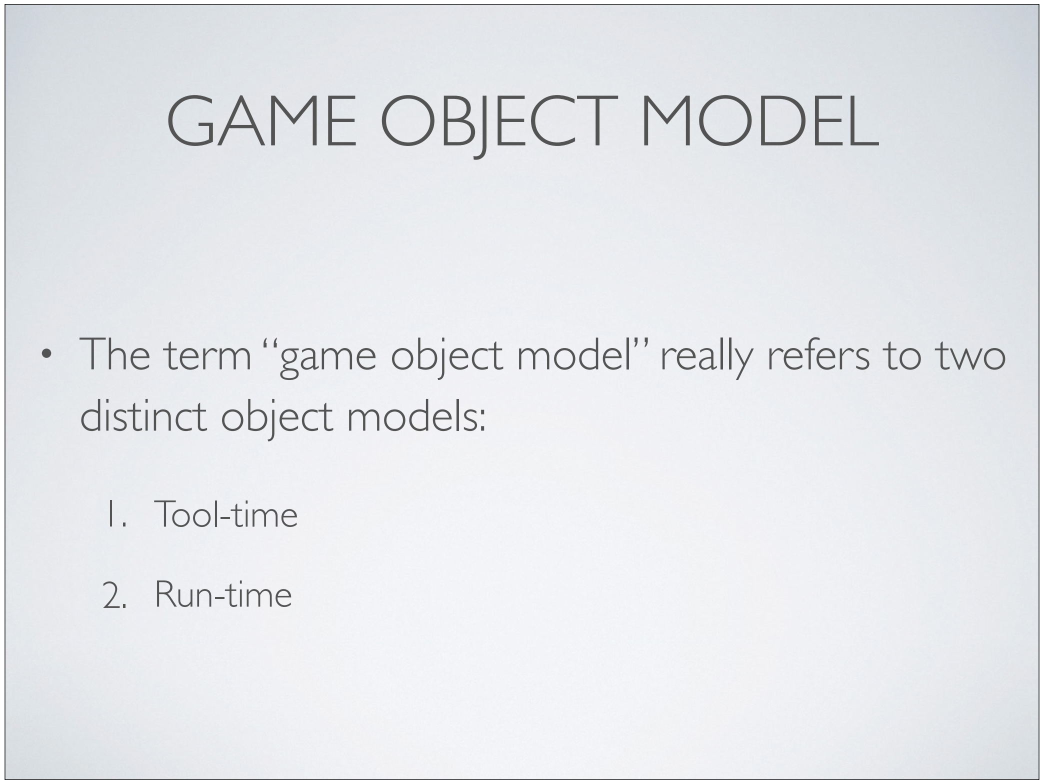
- Two distinct meanings:
    1. Properties/architecture of a particular object-oriented programming language (e.g. C++, Java, Python, …)
    2. Collection of objects/classes with which programs can be built or problems can be solved

1. Properties/architecture of a particular object-oriented programming language (e.g. C++, Java, Python)

* Defines concepts like class, method, inheritance, etc. and specifies implementation details particular to that language

2. Collection of objects/classes with which programs can be built or problems can be solved

* This is what we mean when we talk about a game’s object model

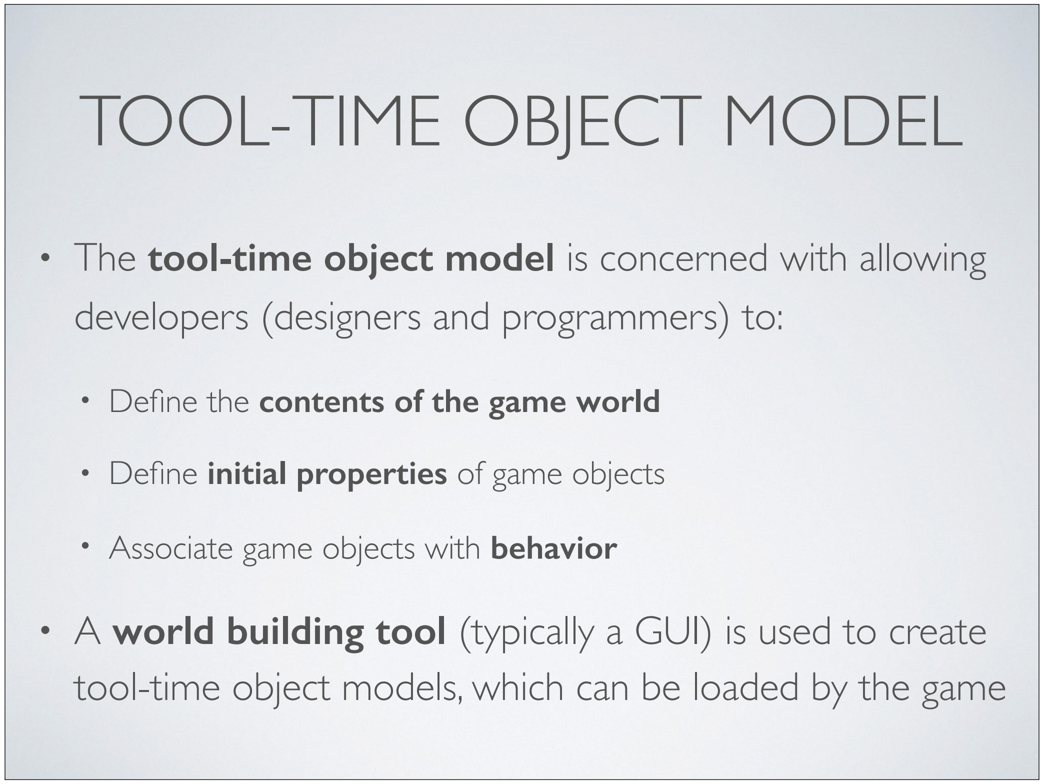


GAME OBJECT MODEL

- The term "game object model" really refers to two distinct object models:
  1. Tool-time
  2. Run-time

The two are closely related conceptually, but the two implementations may differ a great deal

TOOL-TIME OBJECT MODEL

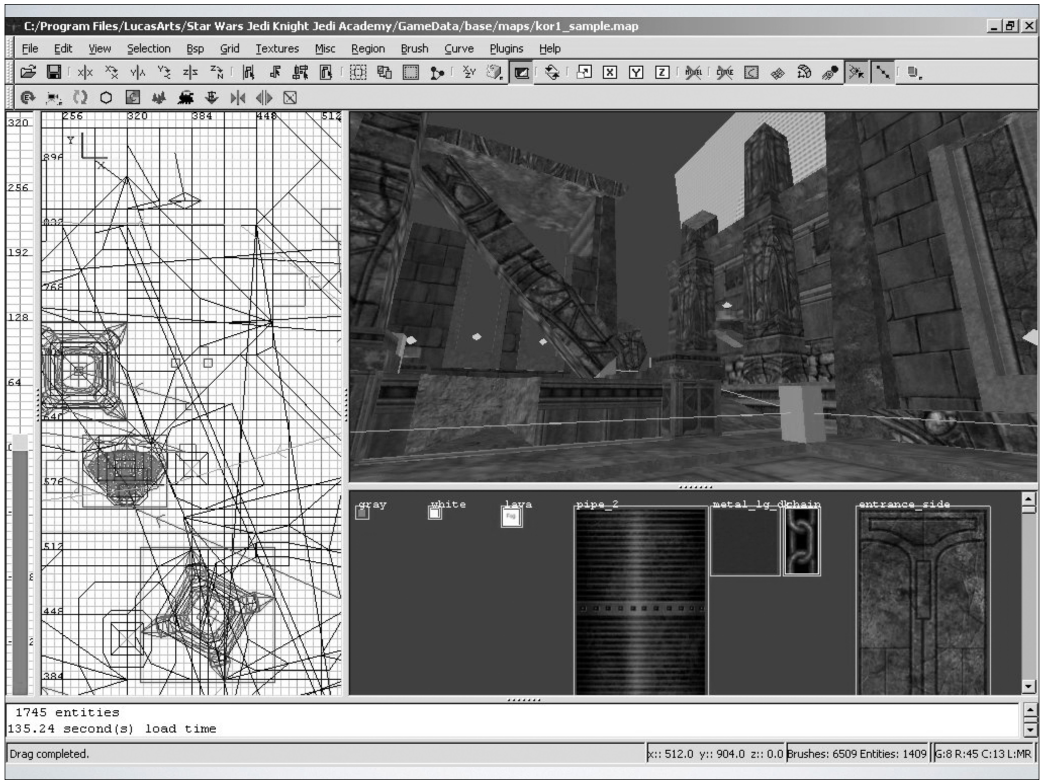


TOOL-TIME OBJECT MODEL

- The **tool-time object model** is concerned with allowing developers (designers and programmers) to:
  - Define the **contents of the game world**
  - Define **initial properties** of game objects
  - Associate game objects with **behavior**
- A **world building tool** (typically a GUI) is used to create tool-time object models, which can be loaded by the game

Define the contents of the game world (**usually on a per-level/per-map basis**)

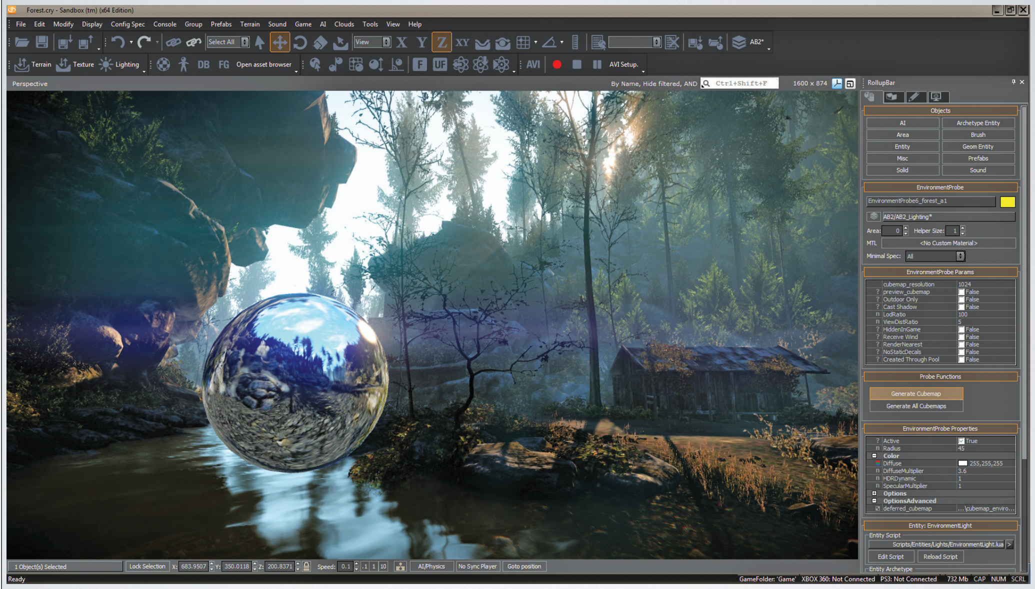
behavior (**hard-coded or scripted**)



C:/Program Files/LucasArts/Star Wars Jedi Knight Jedi Academy/GameData/base/maps/kor1_sample.map

```
 1745 entities
135.24 second(s) load time
```

Drag completed. | x:: 512.0 y:: 904.0 z:: 0.0 | Brushes: 6509 Entities: 1409 | G:8 R:45 C:13 L:MR



EnvironmentProbe

EnvironmentProbe6_forest_a1

AB2/AB2_Lighting*

Area: 0 Helper Size: 1

MTL <No Custom Material>

Minimal Spec: All

EnvironmentProbe Params

cubemap_resolution	1024
preview_cubemap	False
Outdoor Only	False
Cast Shadow	False
LodRatio	100
ViewDistRatio	5
HiddenInGame	False
Receive Wind	False
RenderNearest	False
NoStaticDecals	False
Created Through Pool	False

EnvironmentProbe Properties

Active	True
Radius	45
Color	
Diffuse	255,255,255
DiffuseMultiplier	3.6
HDRDynamic	1
SpecularMultiplier	1
Options	
OptionsAdvanced	
deferred_cubemap	...\cubemap_enviro...

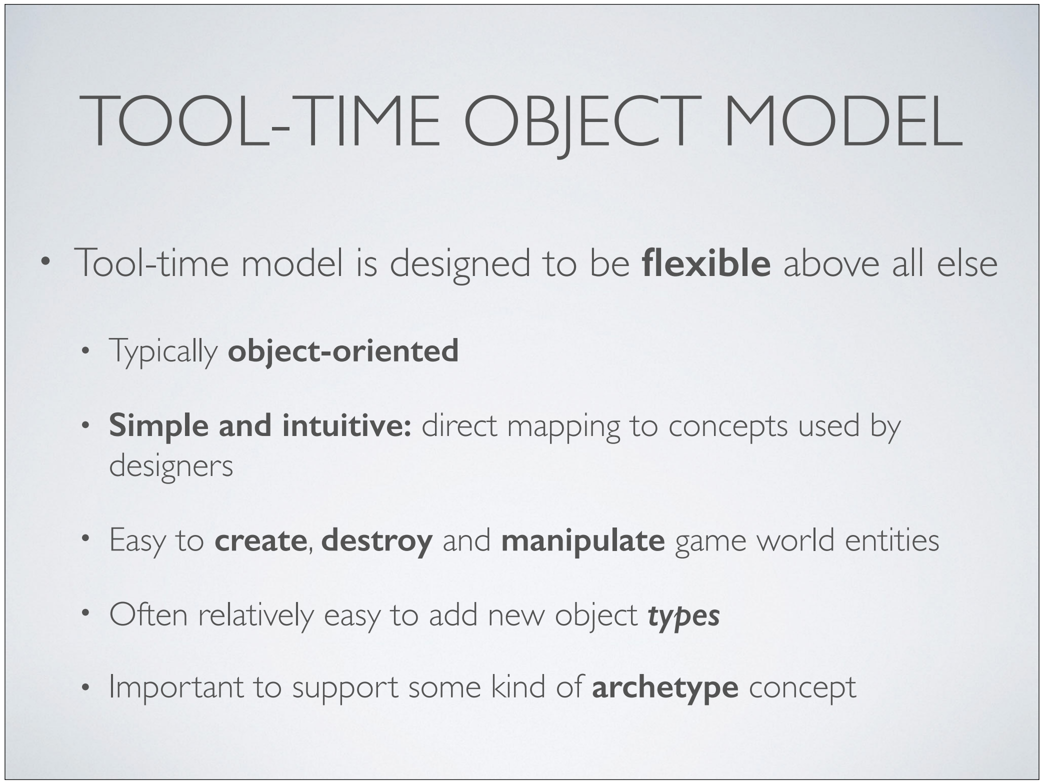
Entity: EnvironmentLight

Entity Script: Scripts/Entities/Lights/EnvironmentLight.lua

Entity Archetype

1 Object(s) Selected X: 683.9507 Y: 350.0118 Z: 200.8371 Speed: 0.1 .1 1 10

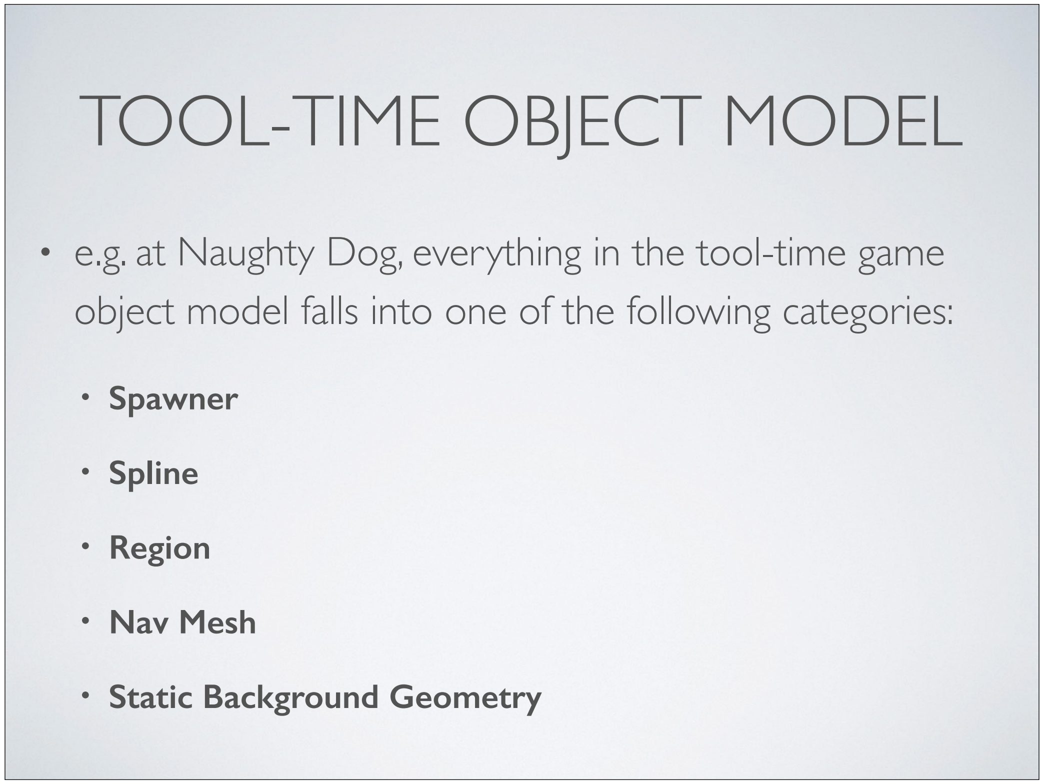
Ready GameFolder: 'Game' XBOX 360: Not Connected PS3: Not Connected 732 Mb CAP NUM SCRL



TOOL-TIME OBJECT MODEL

- Tool-time model is designed to be **flexible** above all else
  - Typically **object-oriented**
  - **Simple and intuitive:** direct mapping to concepts used by designers
  - Easy to **create**, **destroy** and **manipulate** game world entities
  - Often relatively easy to add new object ***types***
  - Important to support some kind of **archetype** concept

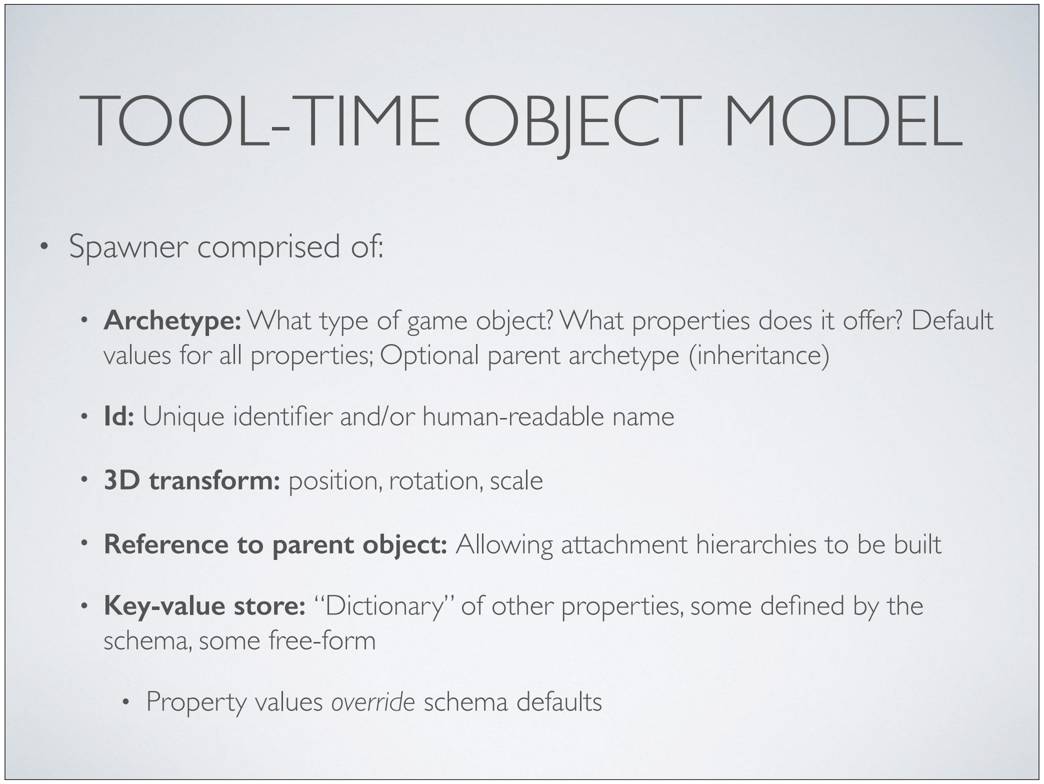
* Typically object-oriented
* Simple and intuitive: direct mapping to concepts used by designers
* Easy to create, destroy and manipulate game world entities
* Often relatively easy to add new object types (may require some programmer support)
* Important to support some kind of archetype concept
   — Default properties for a given “class” of game object, which can be modified to change all instances of that “class” easily, with per-instance overrides
   — Typically supports inheritance



TOOL-TIME OBJECT MODEL

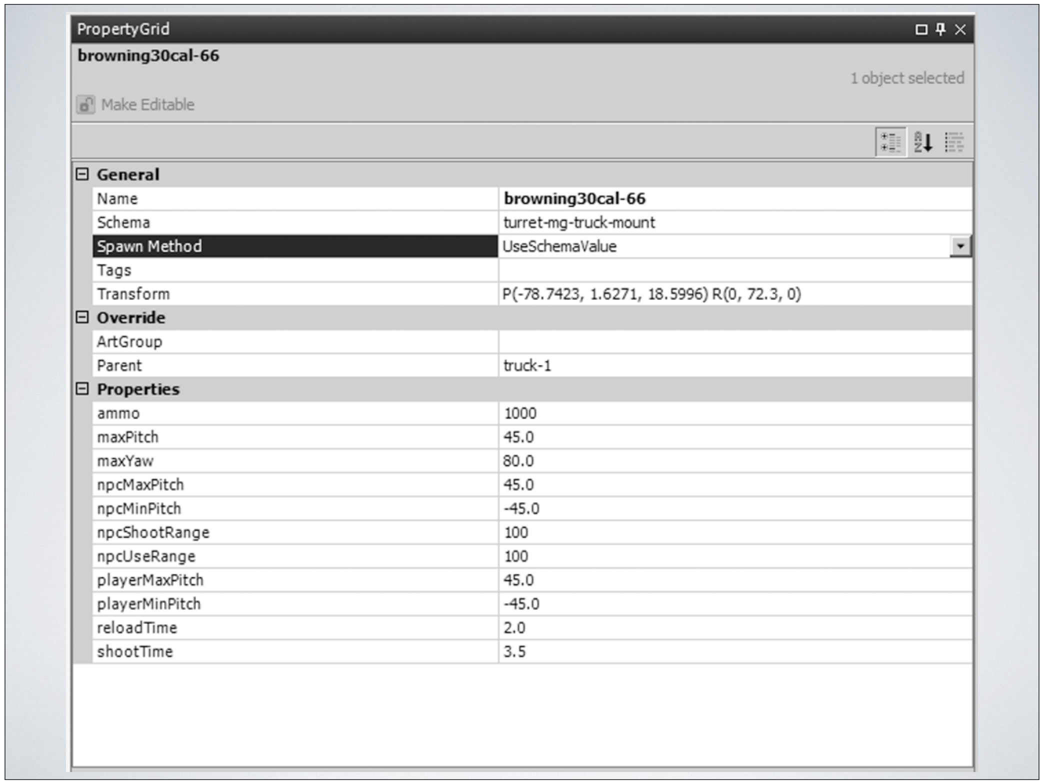
- e.g. at Naughty Dog, everything in the tool-time game object model falls into one of the following categories:
  - **Spawner**
  - **Spline**
  - **Region**
  - **Nav Mesh**
  - **Static Background Geometry**

* Spawner: A key-value store that defines the type of game object, its 3D transform, and its other properties (including optional binding to a script)
* Spline: A set of points/tangents that define a piecewise curve (in our case Catmull-Rom), plus properties which add functionality to the spline
* Region: A 3D volume defined by bounding planes, plus properties which add functionality to each region
* Nav Mesh: A “2.5D” polygon mesh defining “traversable” areas for AI-controlled NPCs
* Static Background Geometry: Buildings, terrain, bridges, etc. (not strictly part of the object model, but part of the “level” format)



TOOL-TIME OBJECT MODEL

- Spawner comprised of:
  - **Archetype:** What type of game object? What properties does it offer? Default values for all properties; Optional parent archetype (inheritance)
  - **Id:** Unique identifier and/or human-readable name
  - **3D transform:** position, rotation, scale
  - **Reference to parent object:** Allowing attachment hierarchies to be built
  - **Key-value store:** "Dictionary" of other properties, some defined by the schema, some free-form
    - Property values *override* schema defaults


PropertyGrid

browning30cal-66

1 object selected

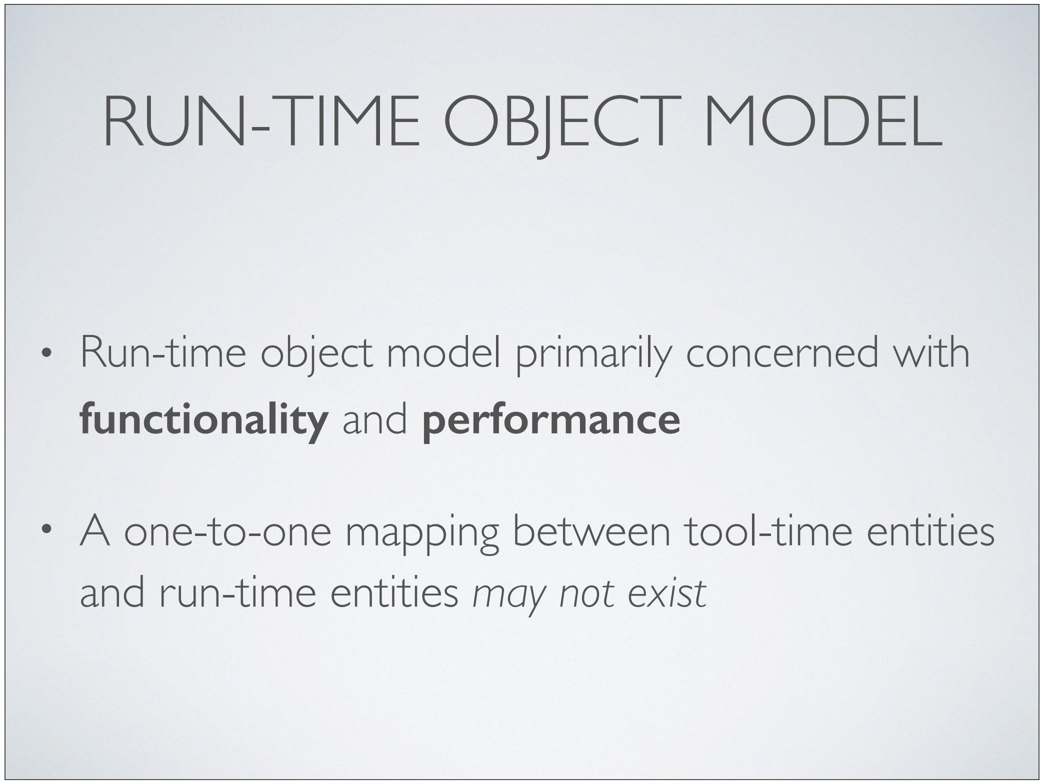
Make Editable

General	
Name	**browning30cal-66**
Schema	turret-mg-truck-mount
Spawn Method	UseSchemaValue
Tags	
Transform	P(-78.7423, 1.6271, 18.5996) R(0, 72.3, 0)
Override	
ArtGroup	
Parent	truck-1
Properties	
ammo	1000
maxPitch	45.0
maxYaw	80.0
npcMaxPitch	45.0
npcMinPitch	-45.0
npcShootRange	100
npcUseRange	100
playerMaxPitch	45.0
playerMinPitch	-45.0
reloadTime	2.0
shootTime	3.5

RUN-TIME OBJECT SYSTEM

RESPONSIBILITIES OF THE RUN-TIME OBJECT SYSTEM

- Representation of the tool-time object model, with behaviors
- Memory management and streaming
- Simulation (updating the model each frame)
- Messaging and event handling
- High-level game flow (objectives, story beats, etc.)
- Scripting (enabling rapid iteration for behavior and flow)

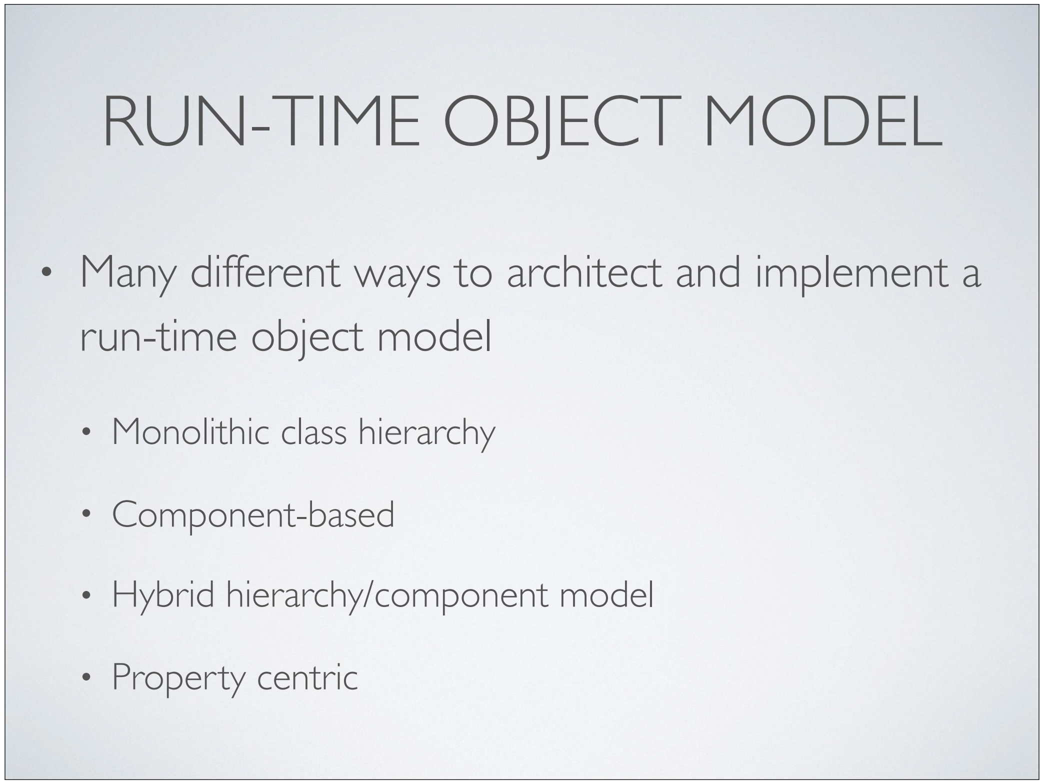


RUN-TIME OBJECT MODEL

- Run-time object model primarily concerned with **functionality** and **performance**
- A one-to-one mapping between tool-time entities and run-time entities *may not exist*

One tool-time entity might be represented by multiple run-time C++ objects

“Array-of-structs” at tool-time could become “struct-of-arrays” at runtime



RUN-TIME OBJECT MODEL

- Many different ways to architect and implement a run-time object model
  - Monolithic class hierarchy
  - Component-based
  - Hybrid hierarchy/component model
  - Property centric

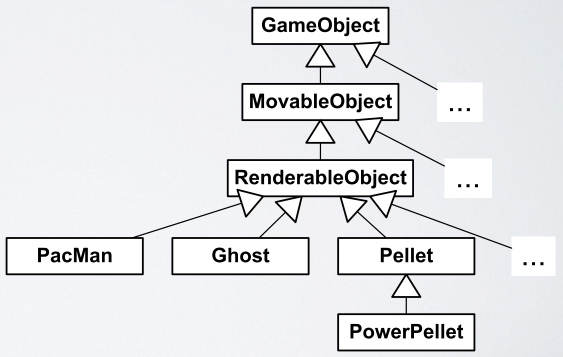
Monolithic class hierarchy (one class per tool-time archetype)

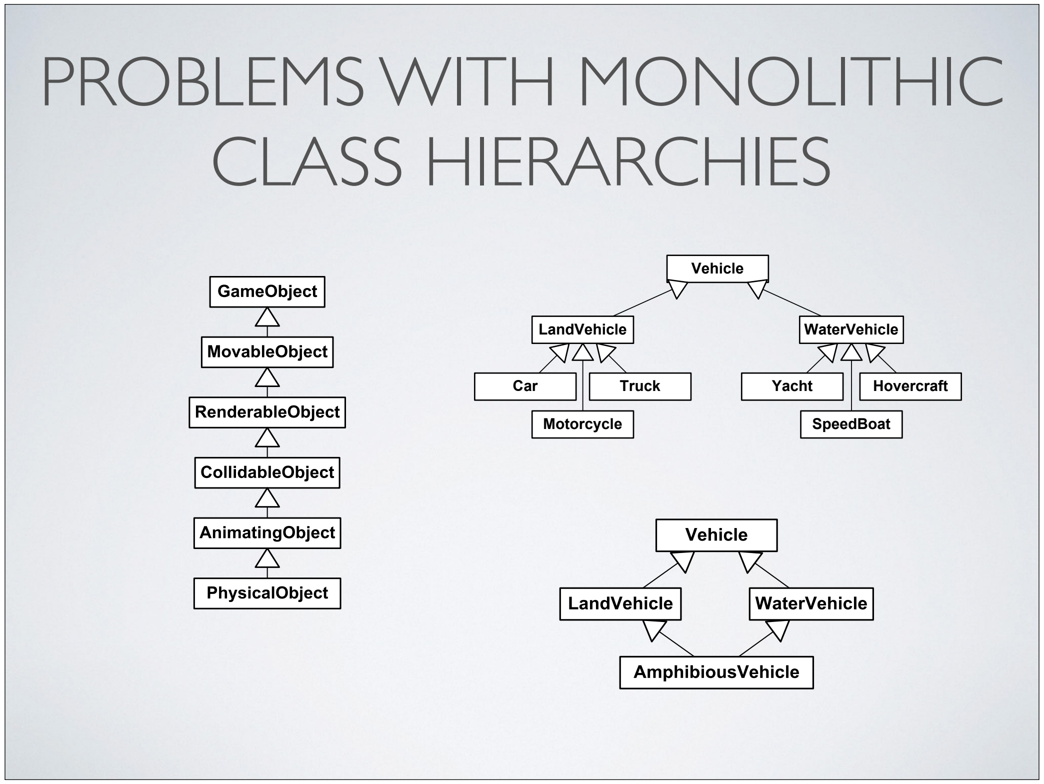
Component-based model (each tool-time entity represented by a collection of run-time component objects)

Property centric: Giant property dictionary!

MONOLITHIC CLASS HIERARCHY

- Obvious choice when run-time language is OOP
- Downsides:
  - Inflexible, hard to maintain
  - Bubble-up problem
  - Deadly diamond problem



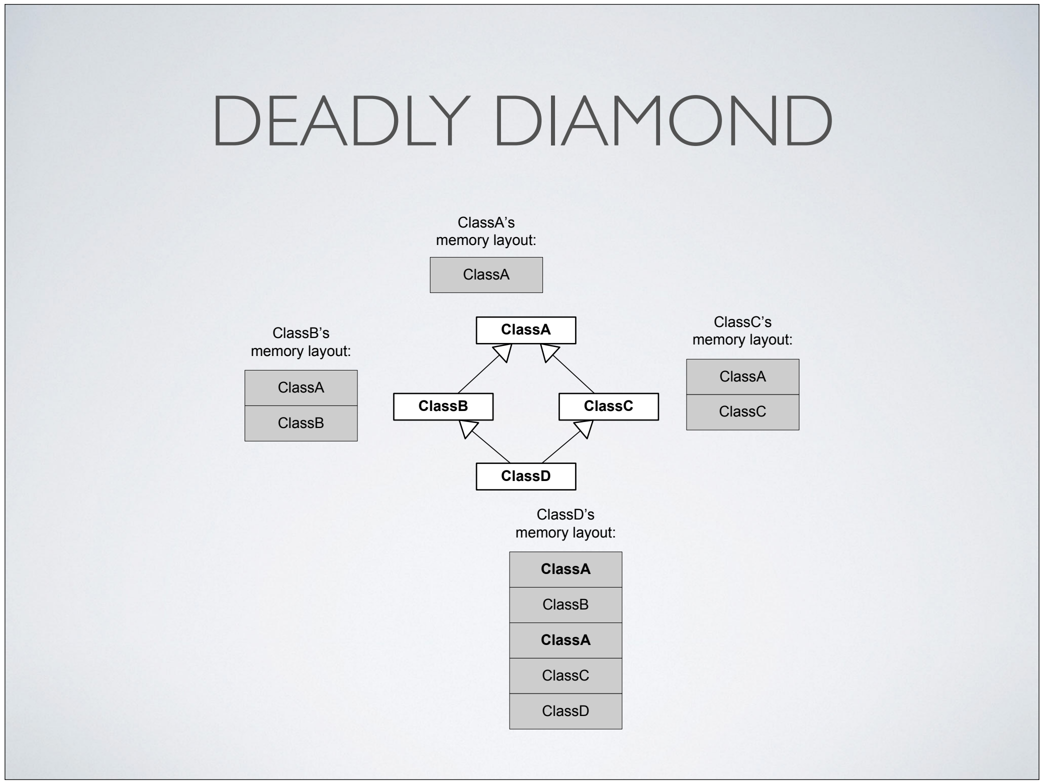


PROBLEMS WITH MONOLITHIC CLASS HIERARCHIES

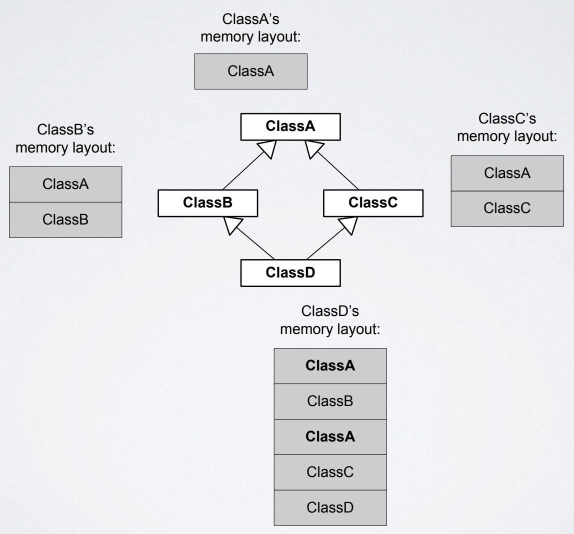
Often not clear which class should be parent of which other class… what if I want an AnimatingObject that has no collision, for example?

Can only categorize along one “axis” at each level of the hierarchy

What seems like a good hierarchy can have a monkey wrench thrown in: AmphibiousVehicle

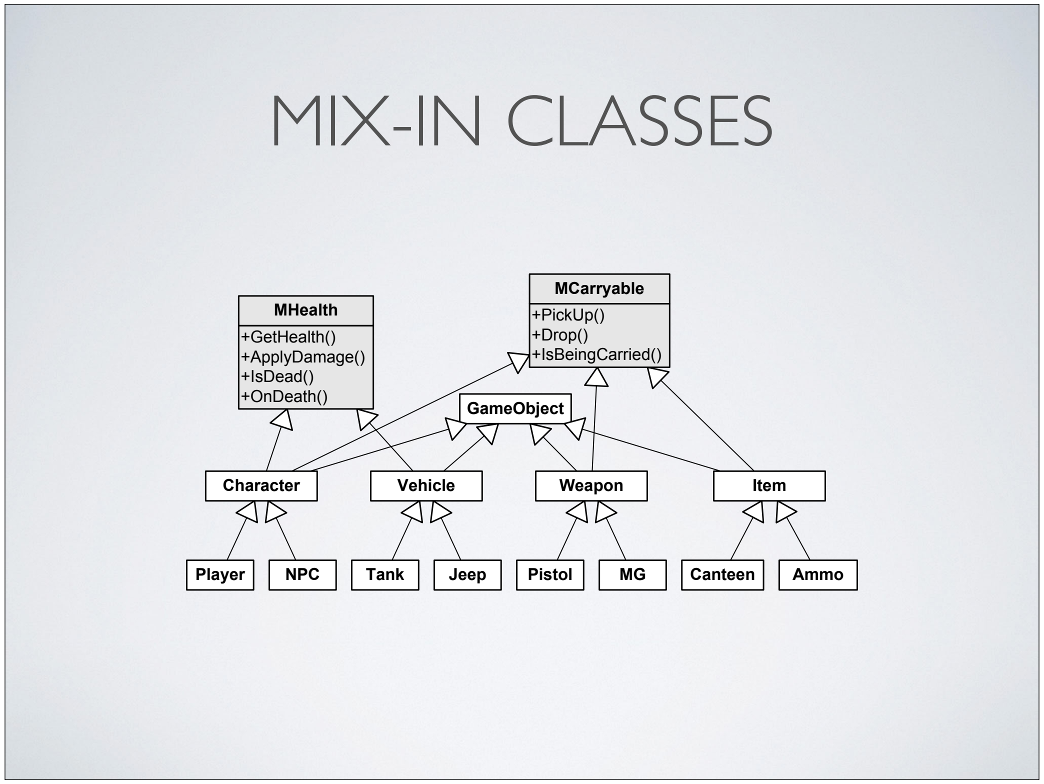


DEADLY DIAMOND

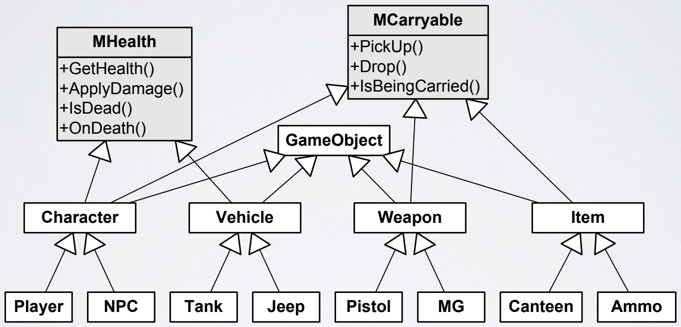


Leads to MI… but MI is problematic for a number of reasons

- deadly diamond: multiple “copies” of a base class, virtual bases
- gets very confusing, difficult to “grok”, hard to maintain
- Multiple __vfptr in object’s memory layout



MIX-IN CLASSES

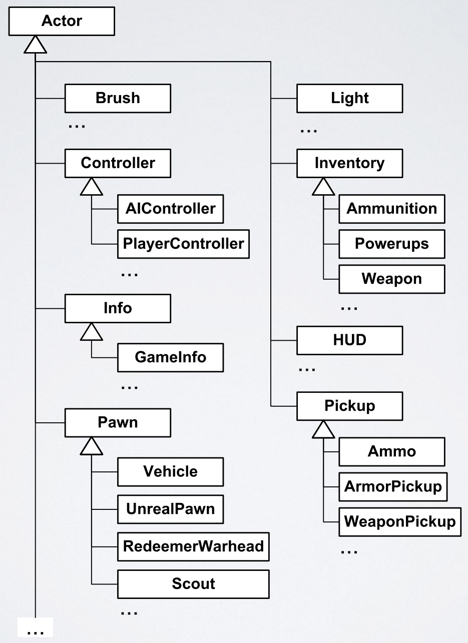


One solution is mix-ins

We do use this here and there at Naughty Dog
* e.g. LinkedNode

Use sparingly (if at all)

THE BUBBLE-UP EFFECT

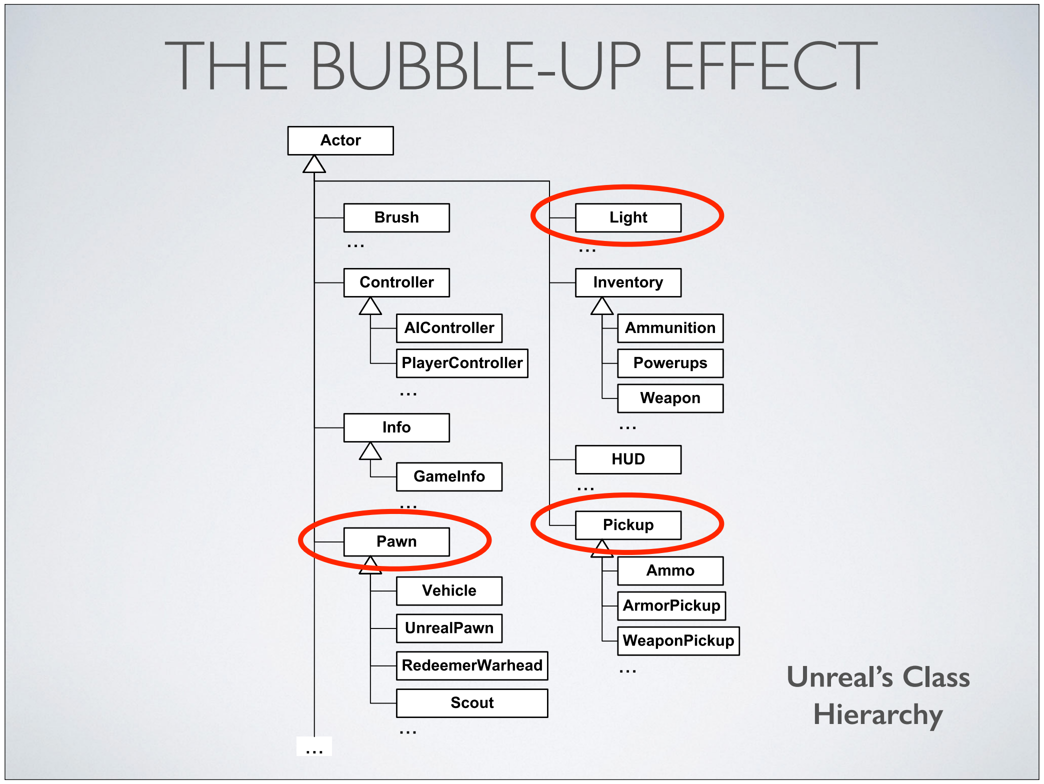


Unreal's Class Hierarchy

this is Unreal's hier — gets complicated pretty fast!
also, suffers from "bubble up" effect:
what if Pawn, Light and Pickup all need a new feature? mix-ins? or just "bubble" that feature up to the only common base class: Actor

as a result, Actor is a giant mess of unrelated features in Unreal

every Actor "pays" for all those features even if it doesn't need them



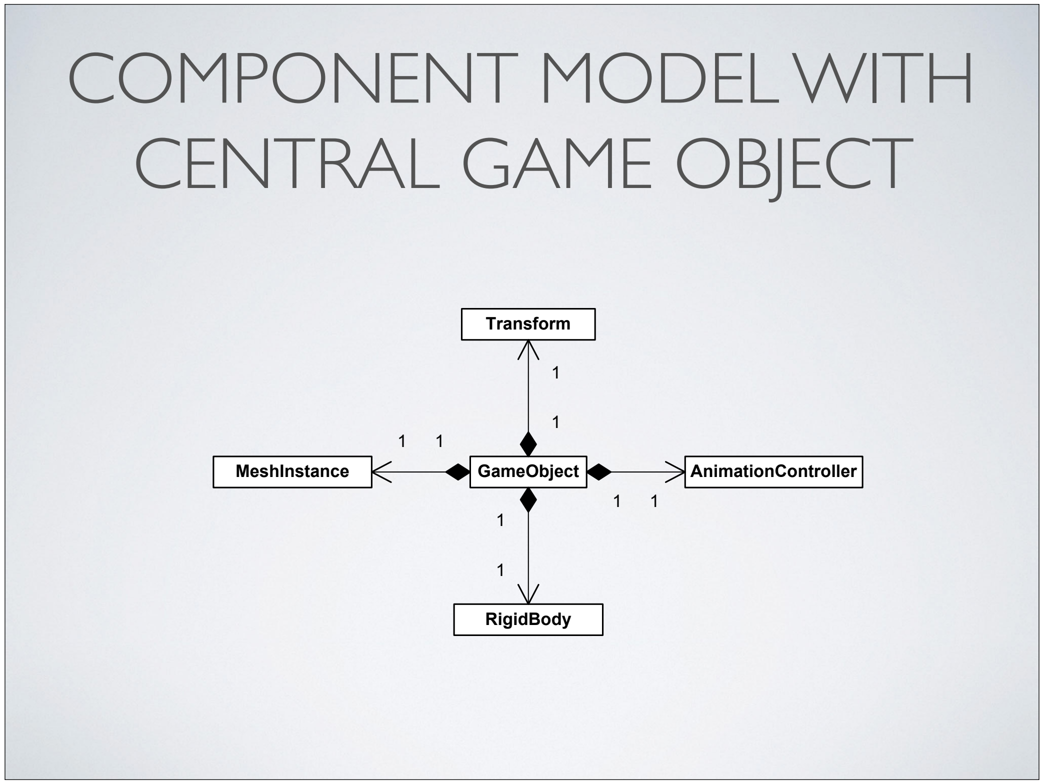
THE BUBBLE-UP EFFECT

Unreal's Class Hierarchy

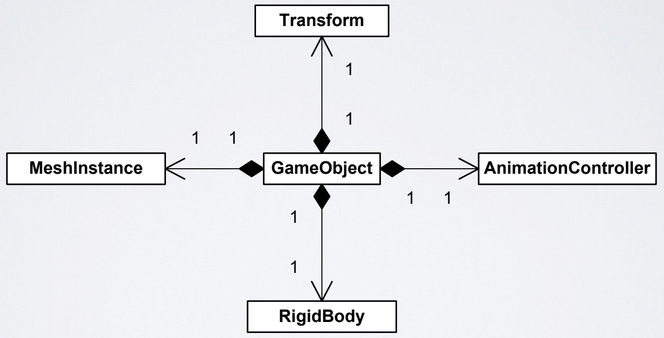
this is Unreal's hier — gets complicated pretty fast!
also, suffers from "bubble up" effect:
what if Pawn, Light and Pickup all need a new feature? mix-ins? or just "bubble" that feature up to the only common base class: Actor

as a result, Actor is a giant mess of unrelated features in Unreal

every Actor "pays" for all those features even if it doesn't need them



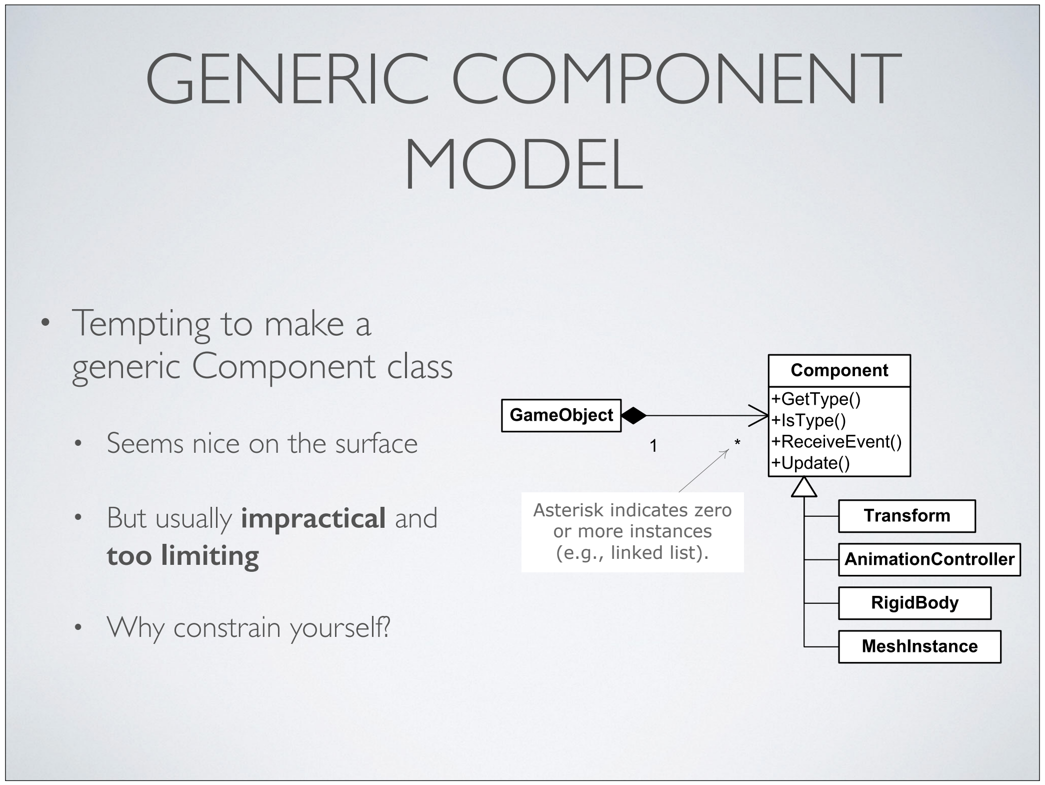
COMPONENT MODEL WITH CENTRAL GAME OBJECT



Can solve a lot of these problems by introducing “components”

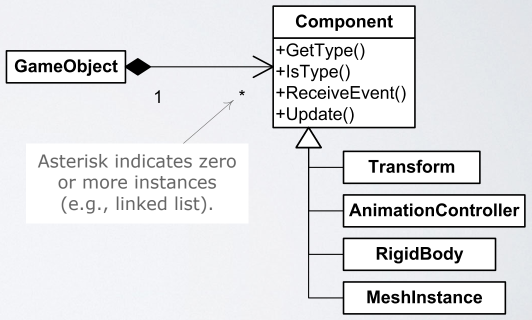
has-a instead of is-a

central game object, with references (strong or weak) to various components



GENERIC COMPONENT MODEL

- Tempting to make a generic Component class
  - Seems nice on the surface
  - But usually **impractical** and **too limiting**
  - Why constrain yourself?

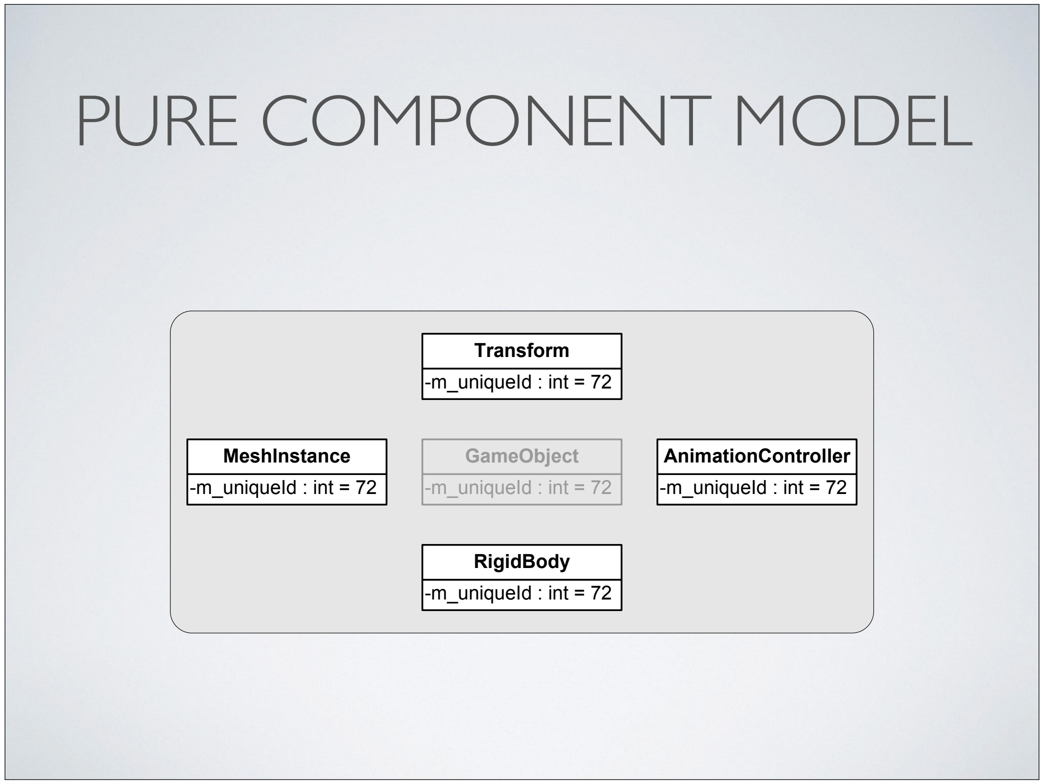


Tempting to make a generic Component class

“Every component MUST BE DERIVED from Component” (or else!)

Seems nice on the surface, but usually impractical and too limiting

Why constrain yourself?

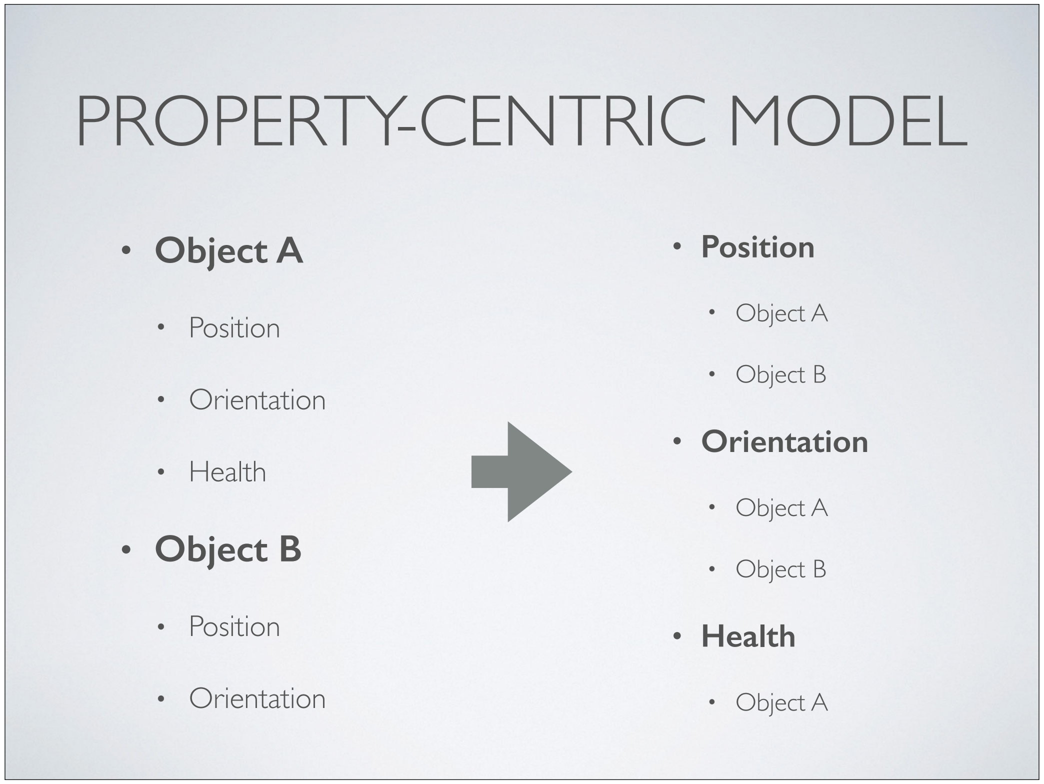


PURE COMPONENT MODEL

why even have the central game object?
just give every component a unique id, and all the ones whose ids match “are” the GO

again, nice idea, seems like it could allow you to “compose” a game object arbitrarily at tool time
but difficult to debug, difficult to work with in practice

Unity does seem to do a nice job of allowing components to be aggregated at tool time (I’d like to know how they get around the impractical aspects, e.g. when it makes no sense to have 5 Transform components, etc.)



PROPERTY-CENTRIC MODEL

- **Object A**
  - Position
  - Orientation
  - Health
- **Object B**
  - Position
  - Orientation

- **Position**
  - Object A
  - Object B
- **Orientation**
  - Object A
  - Object B
- **Health**
  - Object A

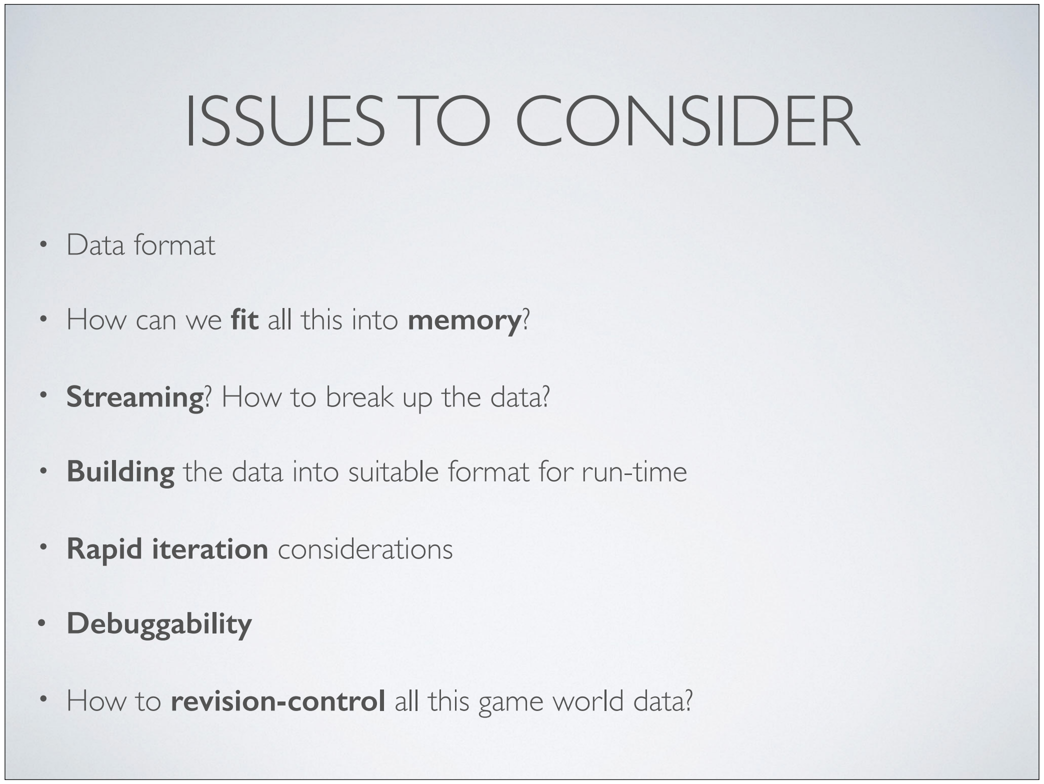
another interesting idea that was used on the game *Thief*
http://chrishecker.com/ images/6/6f/ObjSys.ppt

in my view, costs outweigh benefits… nice to be able to hit a breakpoint and see the object's data in the debugger (but with a debugger add-on that collected and presented all the object's data in one place, I could see it becoming more interesting)

STREAMING AND GAME “FLOW”

LOADING A “MAP”

- Designers create worlds (aka “maps,” aka “levels”) in the world builder tool
- Artists create BG geometry in Maya (or other)
- Need to load this **tool-time object model** into the game
  - Transform into the **run-time object model**!

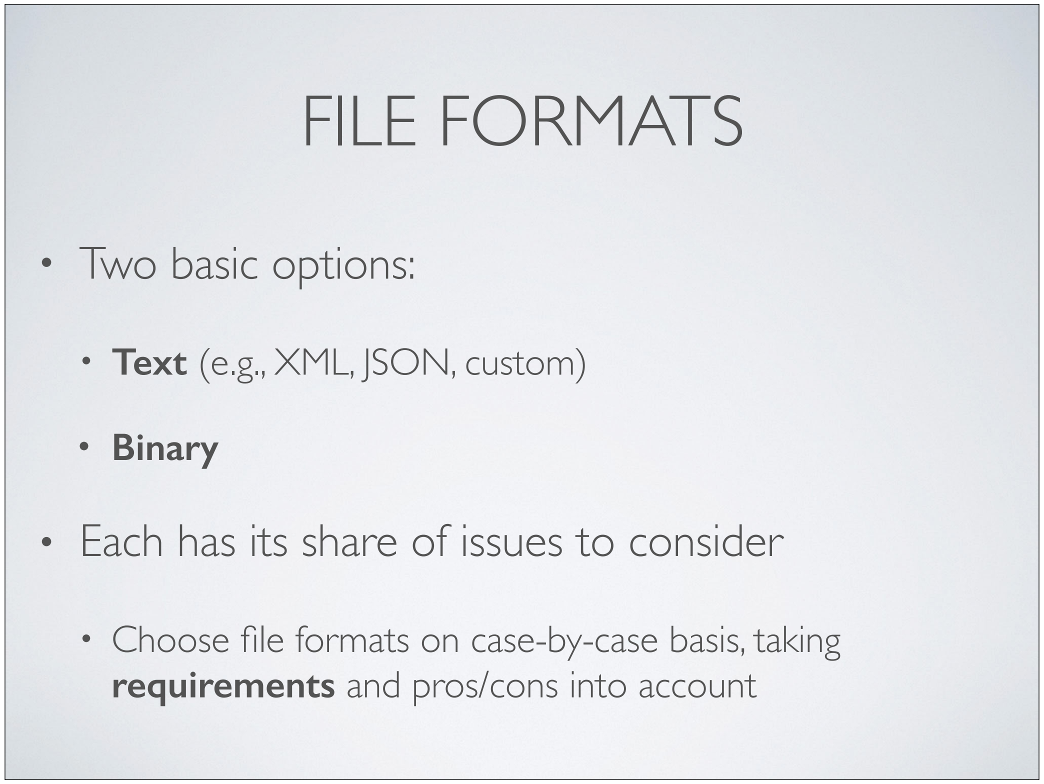


ISSUES TO CONSIDER

- Data format
- How can we **fit** all this into **memory**?
- **Streaming**? How to break up the data?
- **Building** the data into suitable format for run-time
- **Rapid iteration** considerations
- **Debuggability**
- How to **revision-control** all this game world data?

Various issues:

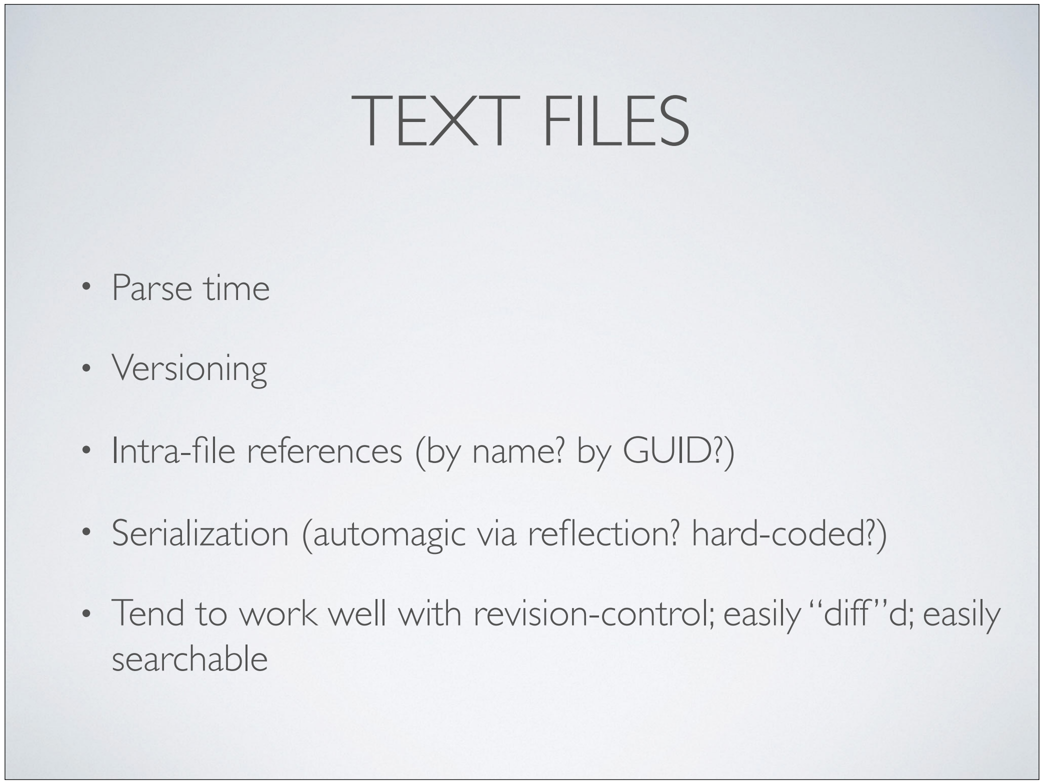
* data format? text? JSON/XML? binary?
* what can fit in memory at any given time? streaming required? (usually)
* how to stream? how break things up so they can stream in efficiently?
* distinction between some data that requires rapid iteration, others that do not
    * in-game pak vs. geometry/skeleton/animation/materials pak
* debuggability: human-readable? searchable data formats?



FILE FORMATS

- Two basic options:
  - **Text** (e.g., XML, JSON, custom)
  - **Binary**
- Each has its share of issues to consider
  - Choose file formats on case-by-case basis, taking **requirements** and pros/cons into account

Important to choose the format on a **case-by-case** basis, according to **requirements**

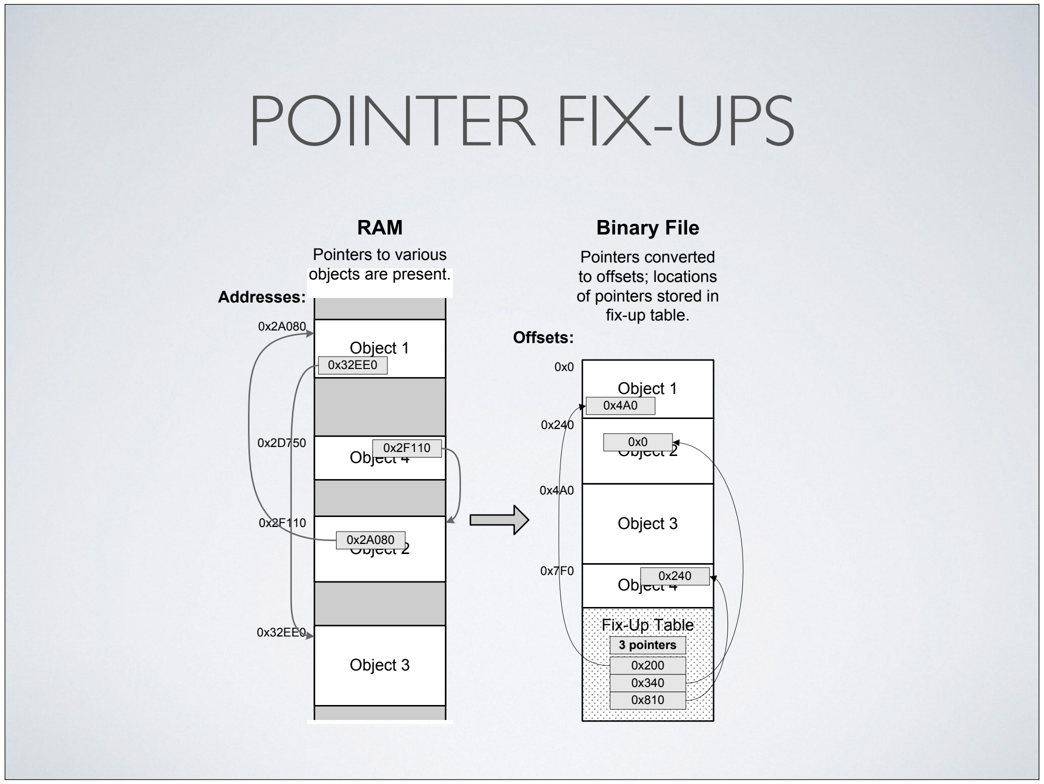

TEXT FILES

- Parse time
- Versioning
- Intra-file references (by name? by GUID?)
- Serialization (automagic via reflection? hard-coded?)
- Tend to work well with revision-control; easily "diff"d; easily searchable

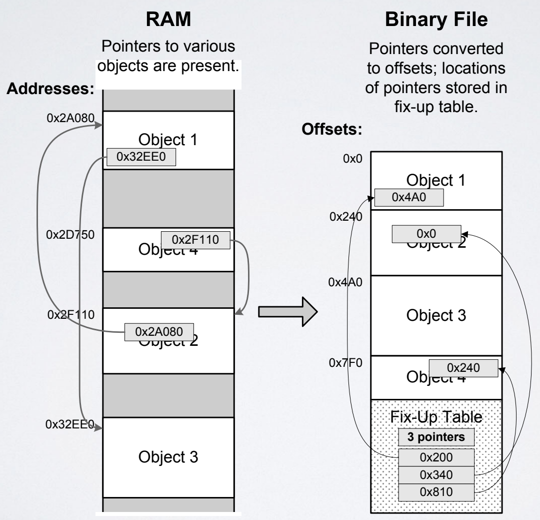
maybe pre-parse a text format into something that can be read more quickly by the engine?

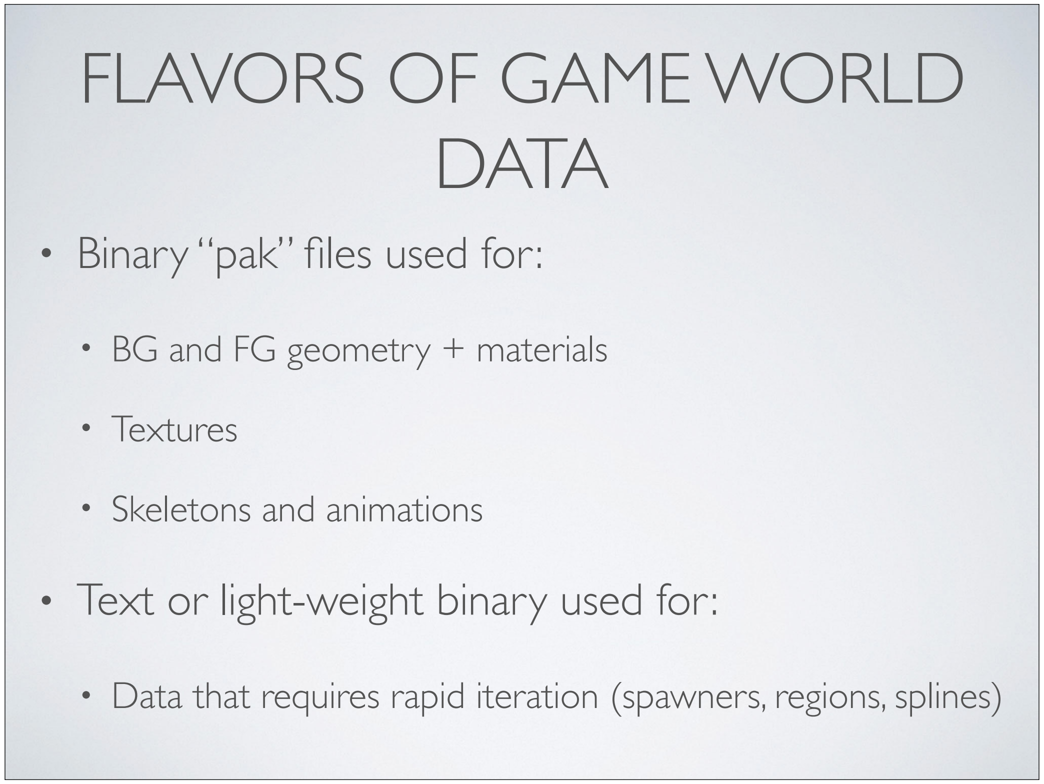
BINARY FILES

- More efficient than text files
- Standard or custom?
- Monolithic or segmented?
- Versioning
- Endian-ness
- Intra-file references (pointer fix-ups)



POINTER FIX-UPS



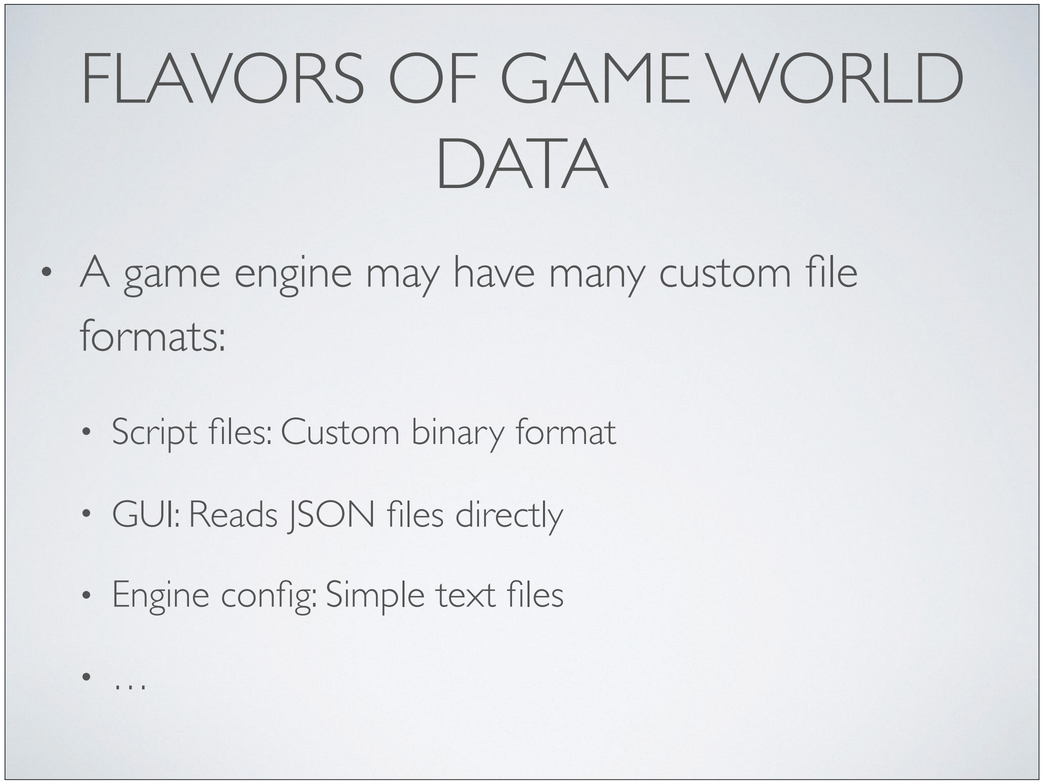


FLAVORS OF GAME WORLD DATA

- Binary "pak" files used for:
  - BG and FG geometry + materials
  - Textures
  - Skeletons and animations
- Text or light-weight binary used for:
  - Data that requires rapid iteration (spawners, regions, splines)

… and may utilize all sorts of other custom formats for specific things
 * script files are stored in their own binary format
 * GUI system uses JSON

This is what we do at Naughty Dog… YMMV



FLAVORS OF GAME WORLD DATA

- A game engine may have many custom file formats:
  - Script files: Custom binary format
  - GUI: Reads JSON files directly
  - Engine config: Simple text files
  - …

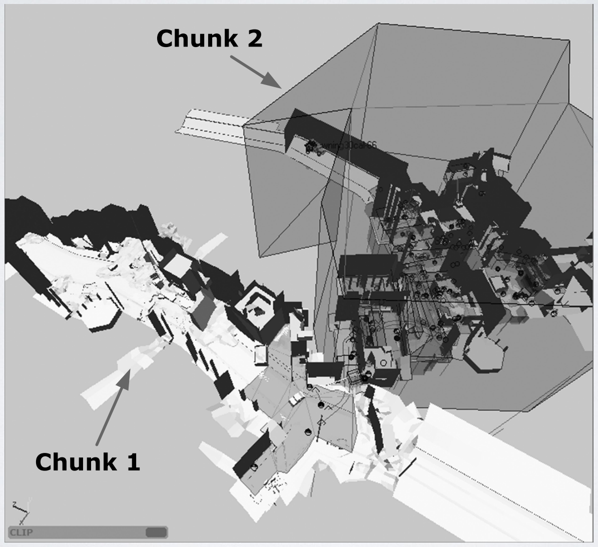
… and may utilize all sorts of other custom formats for specific things
 * script files are stored in their own binary format
 * GUI system uses JSON

This is what we do at Naughty Dog… YMMV

A FEW APPROACHES TO LOADING

- One level at a time (old-school)
- Air locks
- Linear level streaming
  - Related: Streaming audio, animations, textures, …
- LOD-based open world streaming (GTA)

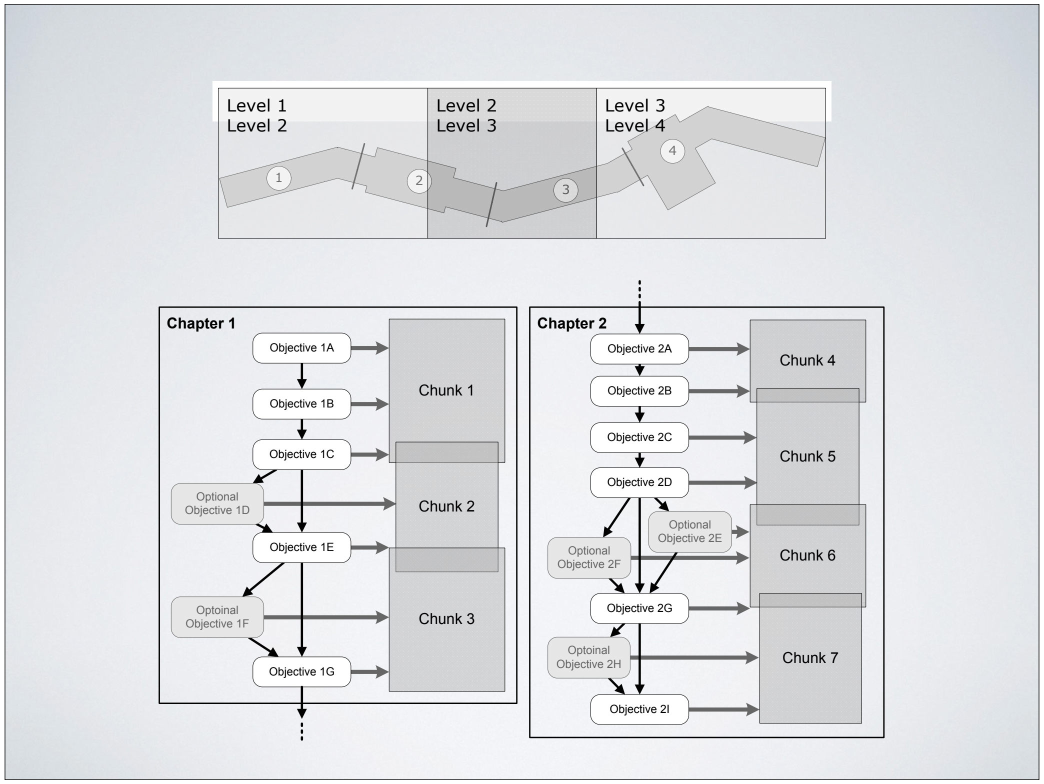
STREAMING



here's how we break up BG geo at Naughty Dog

GTA5 / JustCause do something more sophisticated

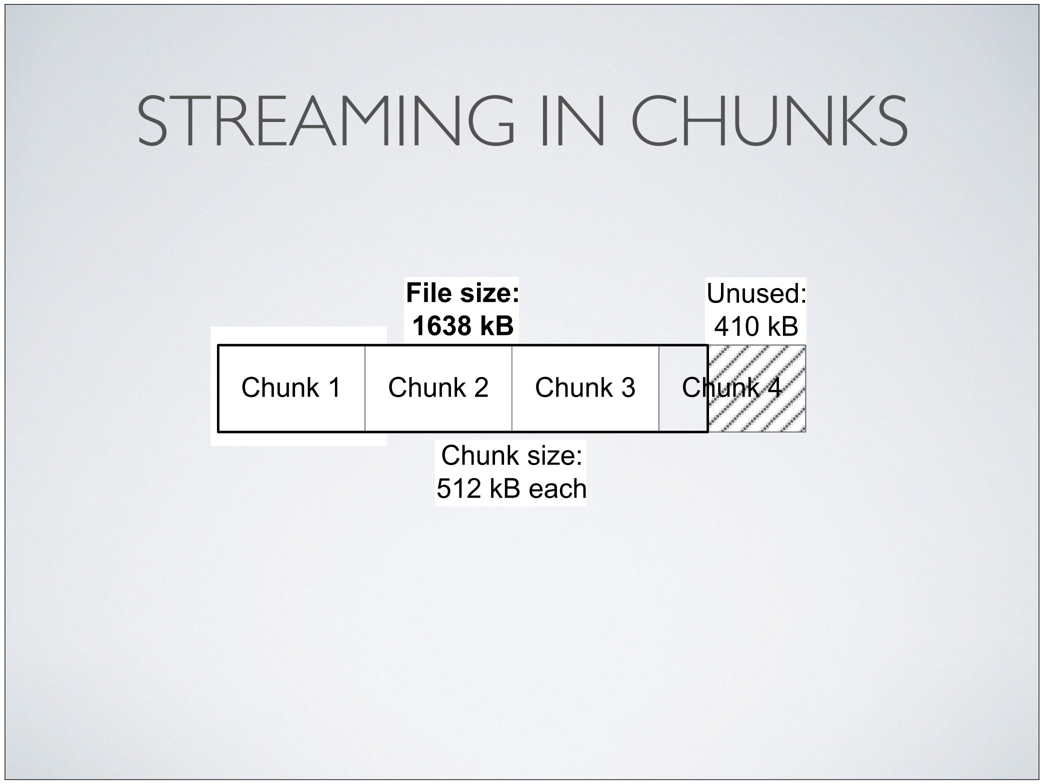
* chunking must be carefully controlled based on geography
* must allow different LODs of the same chunk to be loaded independently



here’s how it might tie in with “objectives” in the game

we have a “task graph” that is the master flow-control; level/chunk loading is driven from it, plus invisible regions in the world that control exactly when each chunk will be added to the “want loads” list, or removed from it

texture streaming is a whole other issue with lots of its own complexity

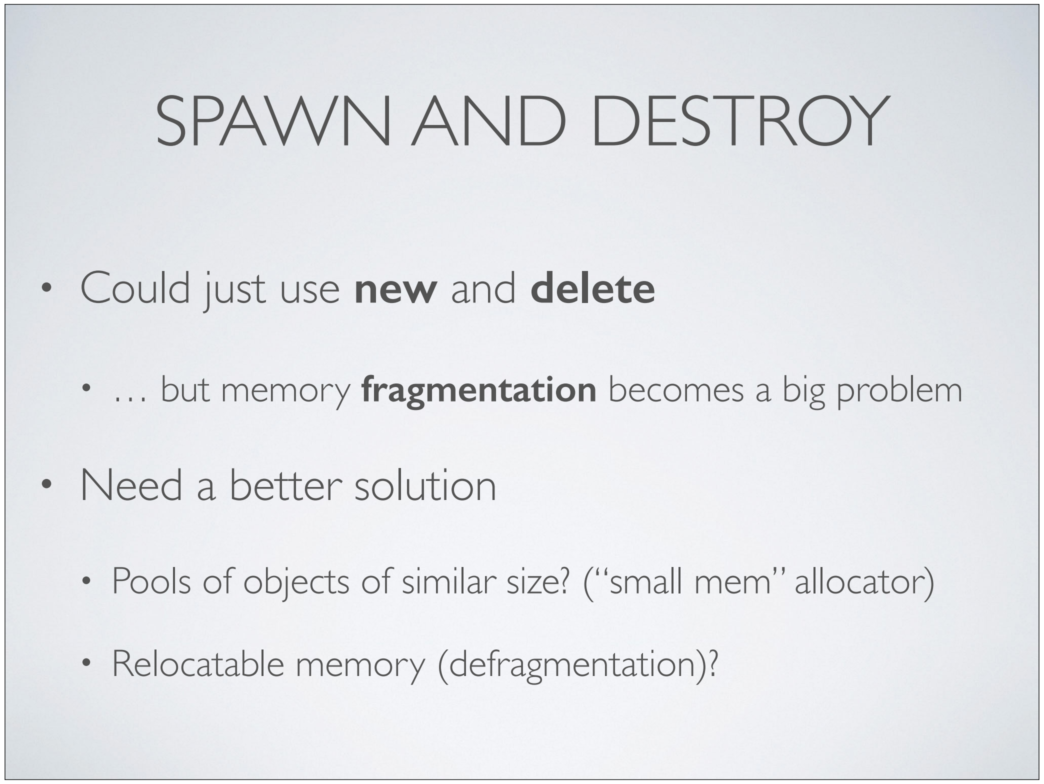


STREAMING IN CHUNKS

we load all data in fixed-sized chunks to reduce the effects of memory fragmentation

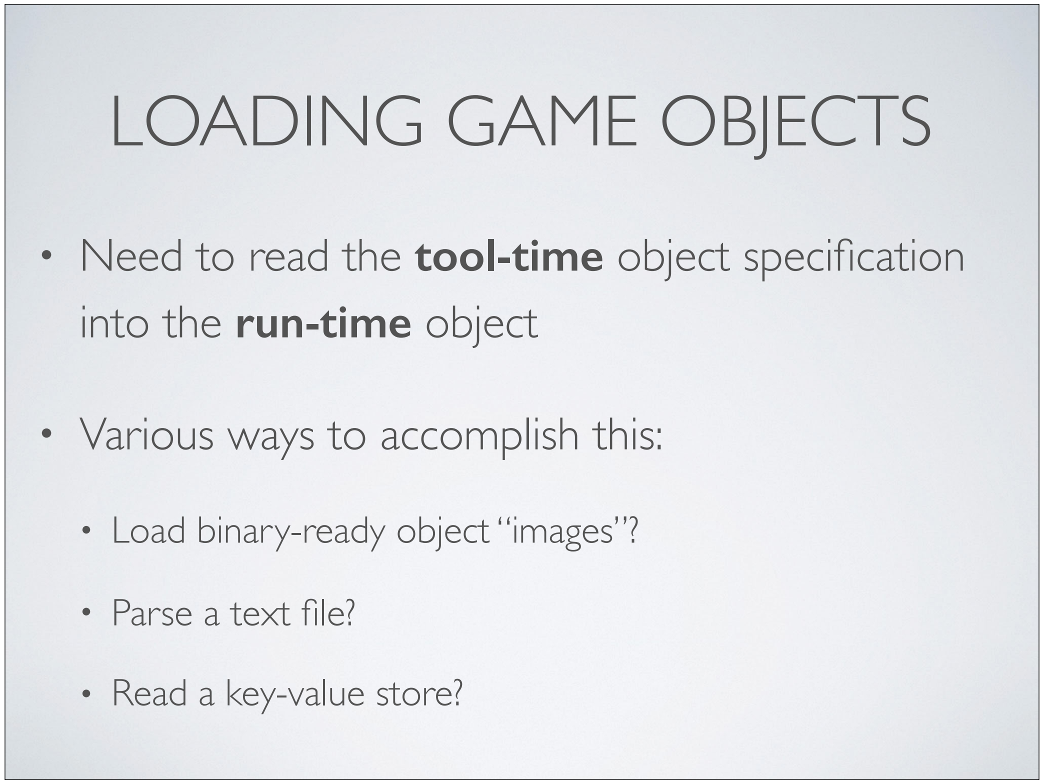
now 1024 KiB actually

SPAWNING AND DESTROYING GAME OBJECTS



SPAWN AND DESTROY

- Could just use **new** and **delete**
  - … but memory **fragmentation** becomes a big problem
- Need a better solution
  - Pools of objects of similar size? (“small mem” allocator)
  - Relocatable memory (defragmentation)?



LOADING GAME OBJECTS

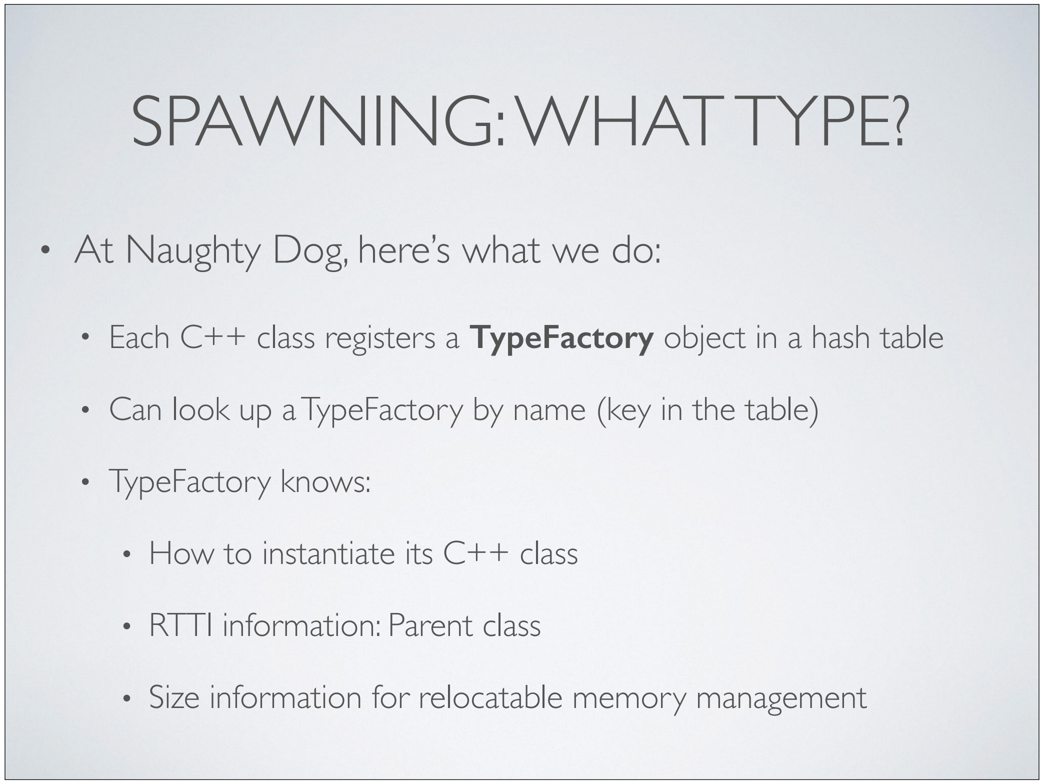
- Need to read the **tool-time** object specification into the **run-time** object
- Various ways to accomplish this:
  - Load binary-ready object "images"?
  - Parse a text file?
  - Read a key-value store?

At NDI we have the concept of a spawner, which is really just a k-v store

Spawning

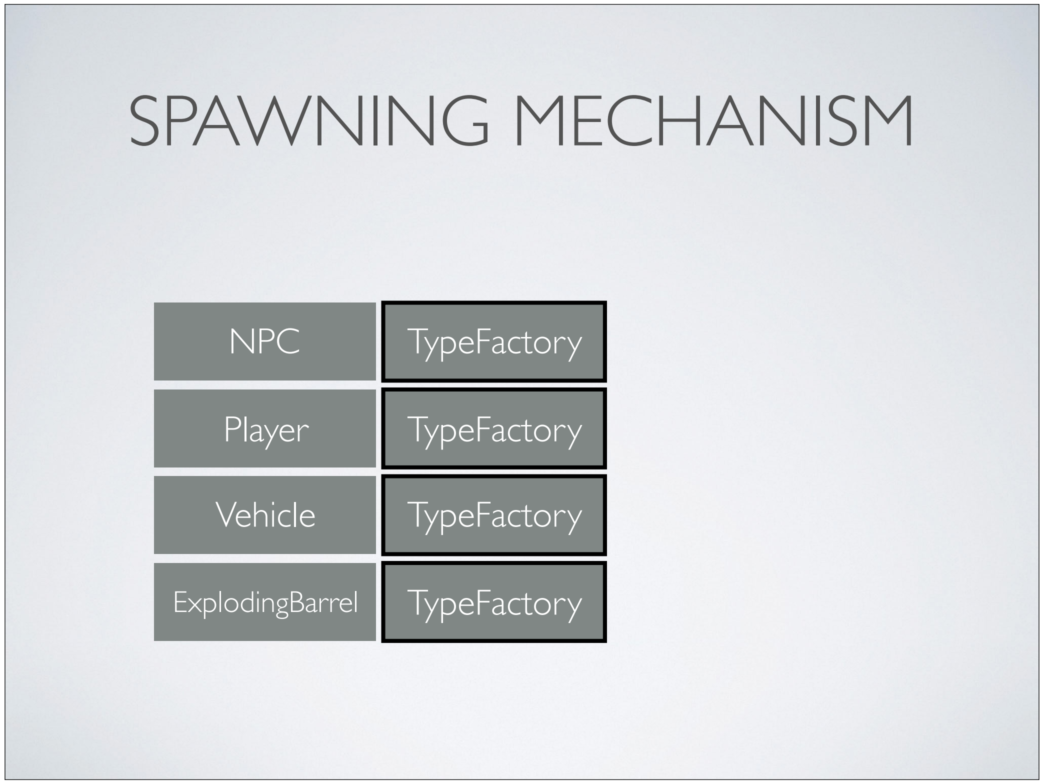
SPAWNING: WHAT TYPE?

- What type of game object(s) to create?
- Need a way to turn tool-time **type descriptor** into an **instance** of the appropriate C++ class(es) at run-time



SPAWNING: WHAT TYPE?

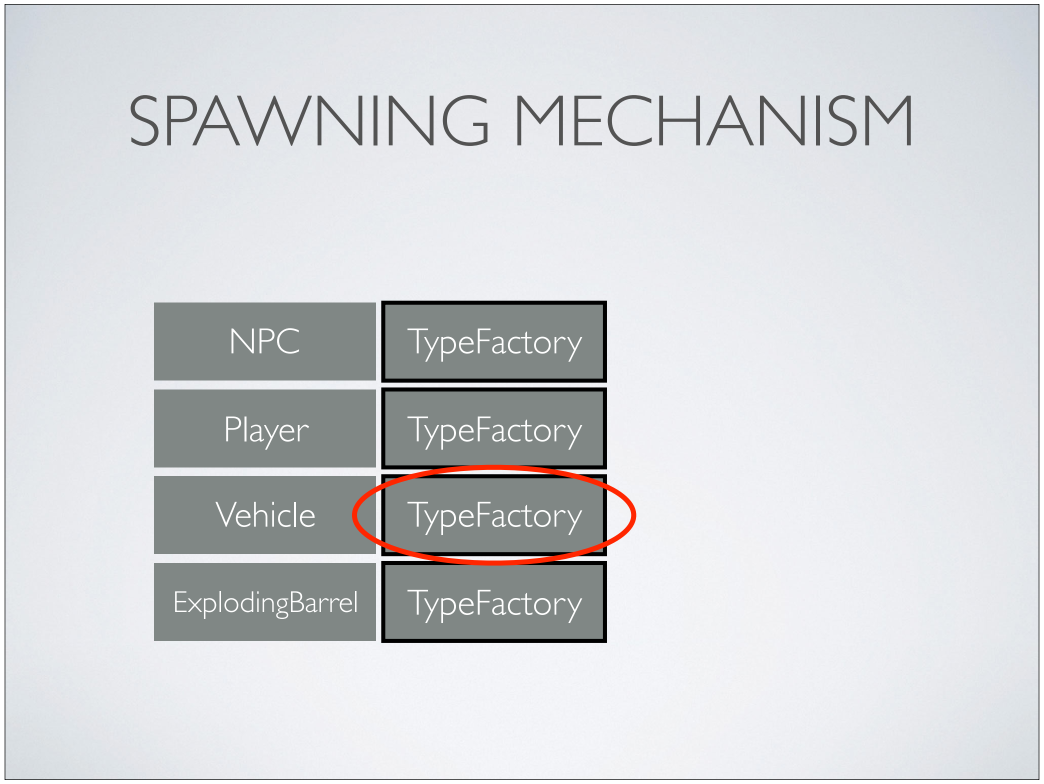
- At Naughty Dog, here's what we do:
  - Each C++ class registers a **TypeFactory** object in a hash table
  - Can look up a TypeFactory by name (key in the table)
  - TypeFactory knows:
    - How to instantiate its C++ class
    - RTTI information: Parent class
    - Size information for relocatable memory management



SPAWNING MECHANISM

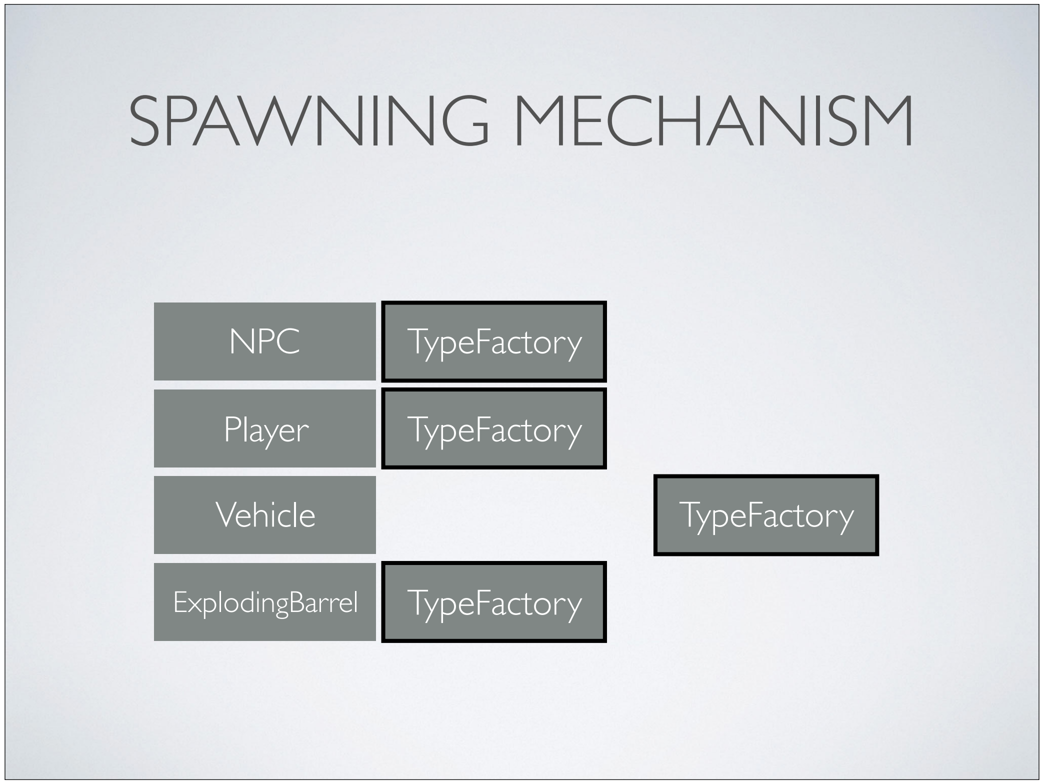
NPC	TypeFactory
Player	TypeFactory
Vehicle	TypeFactory
ExplodingBarrel	TypeFactory

Spawning an object involves looking up the TypeFactory,
asking it for the block size to allocate (max size),
allocating a block in relocatable heap
instantiate via placement new
set up a stack-based allocator within the block for use by the class
call Init(), pass the Spawner so it can read the data store and init all run-time data members
“shrink” the block to fit what was actually used



SPAWNING MECHANISM

Spawning an object involves looking up the TypeFactory,
asking it for the block size to allocate (max size),
allocating a block in relocatable heap
instantiate via placement new
set up a stack-based allocator within the block for use by the class
call Init(), pass the Spawner so it can read the data store and init all run-time data members
“shrink” the block to fit what was actually used

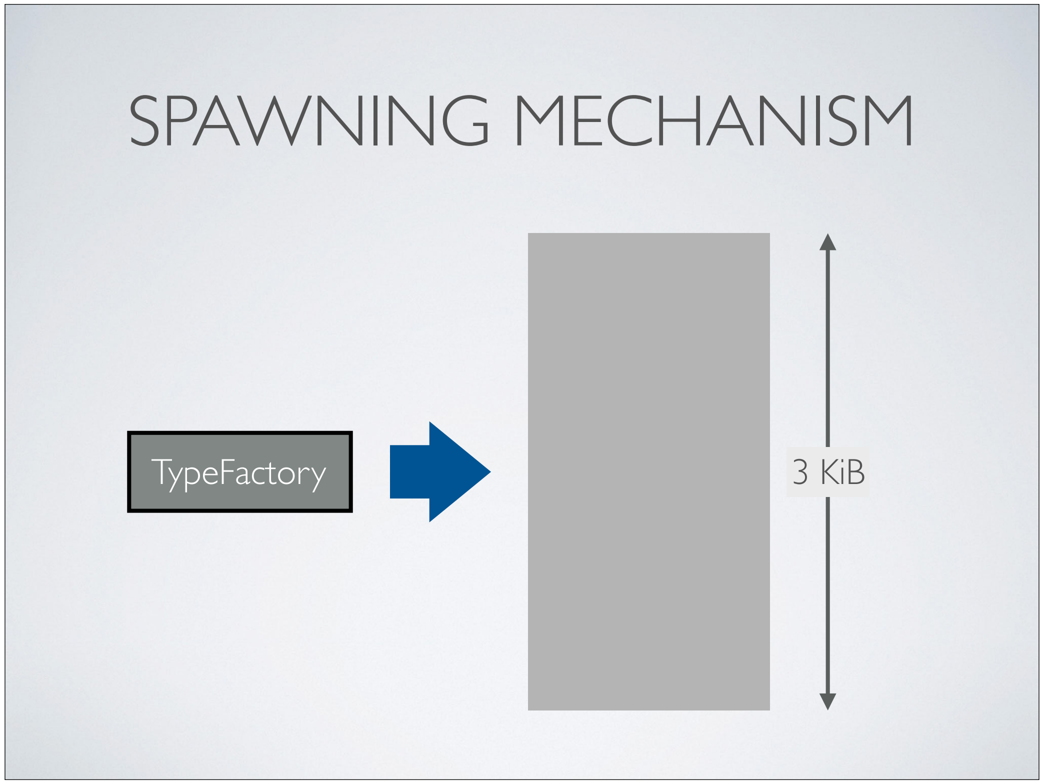


SPAWNING MECHANISM

Spawning an object involves looking up the TypeFactory,
asking it for the block size to allocate (max size),
allocating a block in relocatable heap
instantiate via placement new
set up a stack-based allocator within the block for use by the class
call Init(), pass the Spawner so it can read the data store and init all run-time data members
“shrink” the block to fit what was actually used

SPAWNING MECHANISM

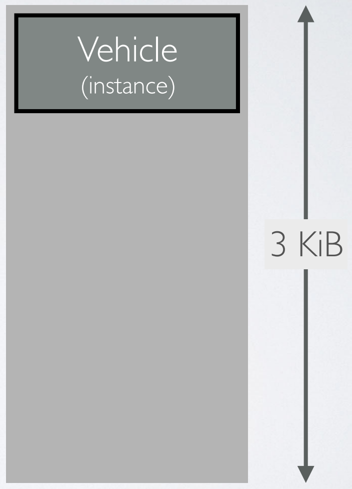
Spawning an object involves looking up the TypeFactory,
asking it for the block size to allocate (max size),
allocating a block in relocatable heap
instantiate via placement new
set up a stack-based allocator within the block for use by the class
call Init(), pass the Spawner so it can read the data store and init all run-time data members
…



SPAWNING MECHANISM

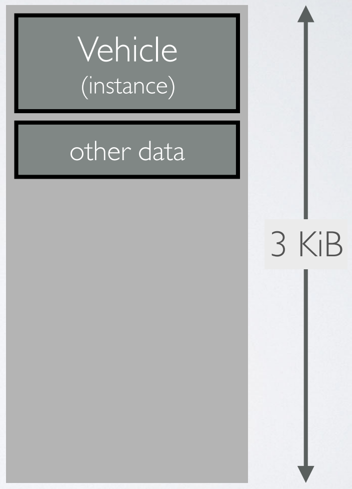
Spawning an object involves looking up the TypeFactory,
asking it for the block size to allocate (max size),
allocating a block in relocatable heap
instantiate via placement new
set up a stack-based allocator within the block for use by the class
call Init(), pass the Spawner so it can read the data store and init all run-time data members
…

SPAWNING MECHANISM

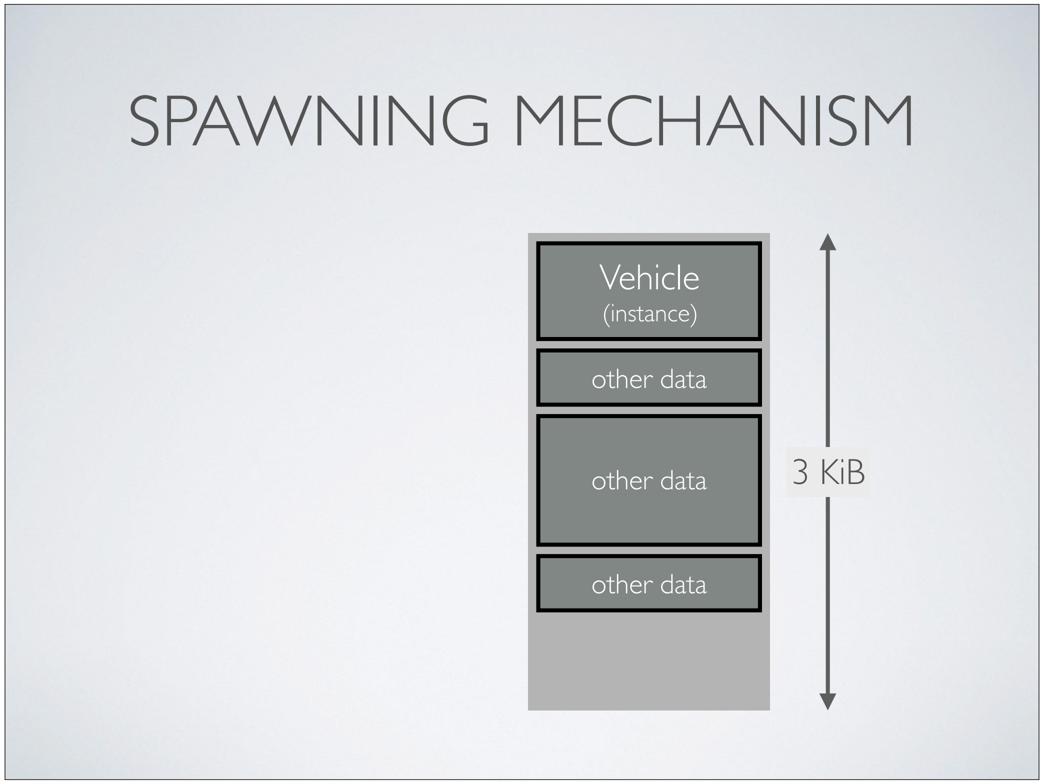


Spawning an object involves looking up the TypeFactory,
asking it for the block size to allocate (max size),
allocating a block in relocatable heap
instantiate via placement new
set up a stack-based allocator within the block for use by the class
call Init(), pass the Spawner so it can read the data store and init all run-time data members
…

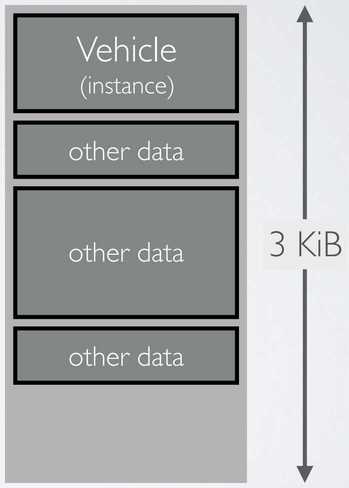
SPAWNING MECHANISM



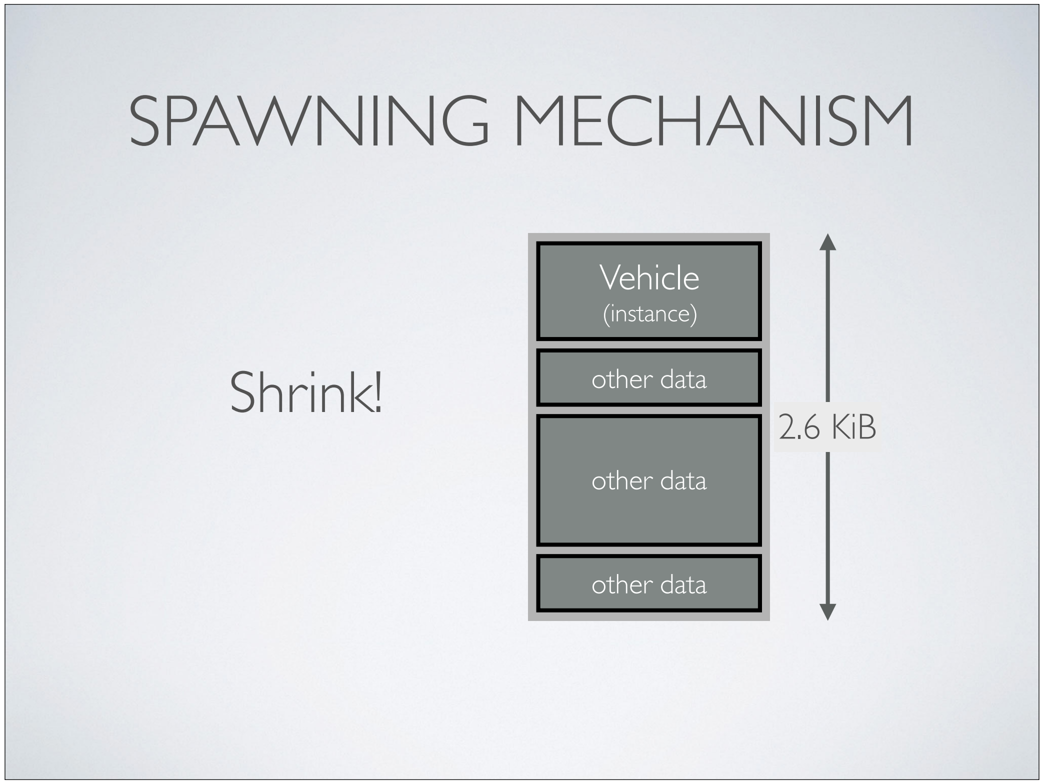
Spawning an object involves looking up the TypeFactory,
asking it for the block size to allocate (max size),
allocating a block in relocatable heap
instantiate via placement new
set up a stack-based allocator within the block for use by the class
call Init(), pass the Spawner so it can read the data store and init all run-time data members
…



SPAWNING MECHANISM

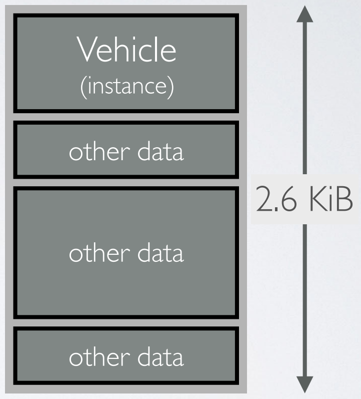


Spawning an object involves looking up the TypeFactory,
asking it for the block size to allocate (max size),
allocating a block in relocatable heap
instantiate via placement new
set up a stack-based allocator within the block for use by the class
call Init(), pass the Spawner so it can read the data store and init all run-time data members
…



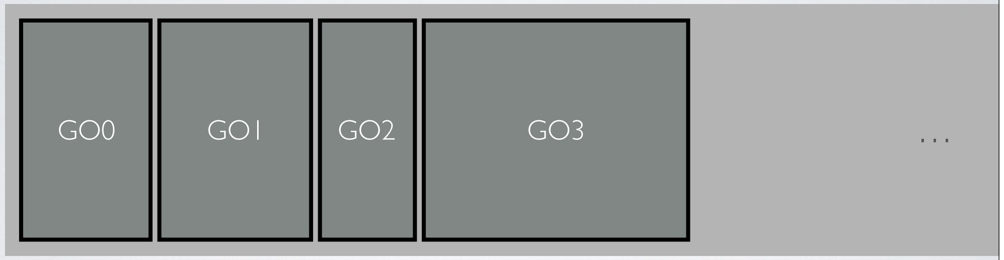
SPAWNING MECHANISM

Shrink!

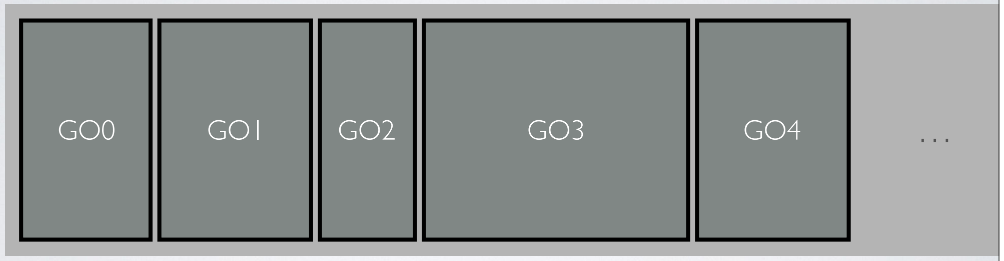


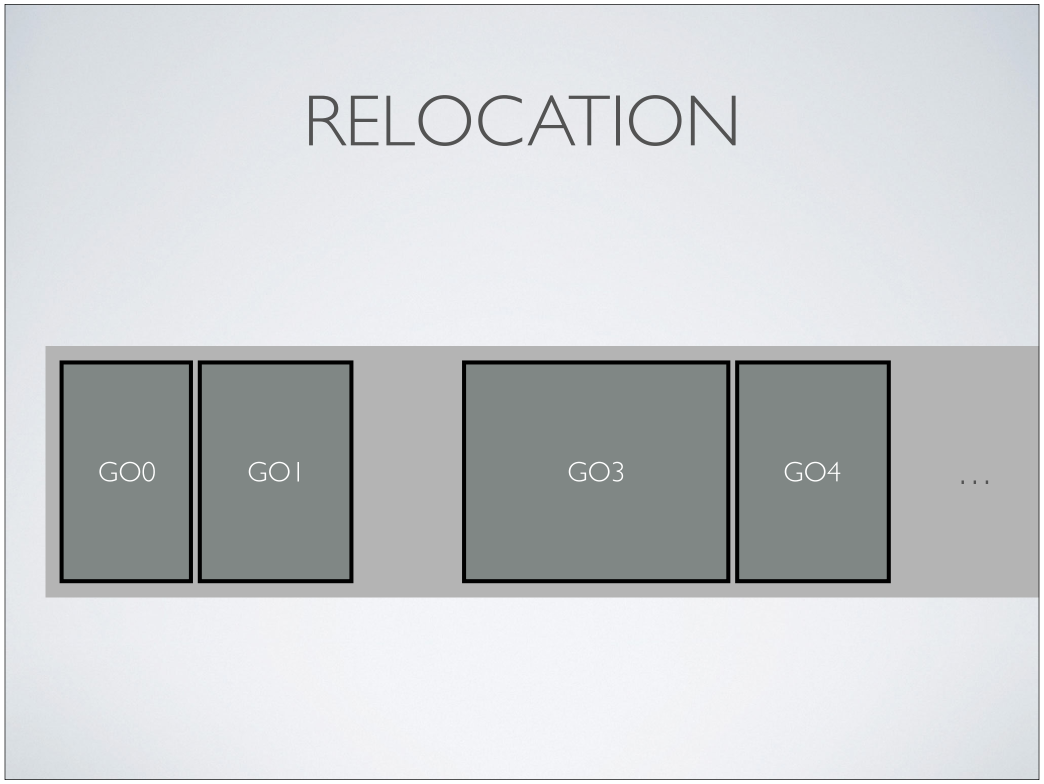
finally, “shrink” the block to fit what was actually used

RELOCATION



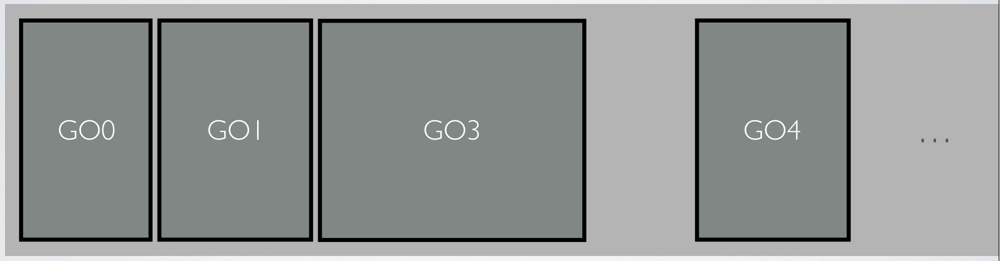
RELOCATION



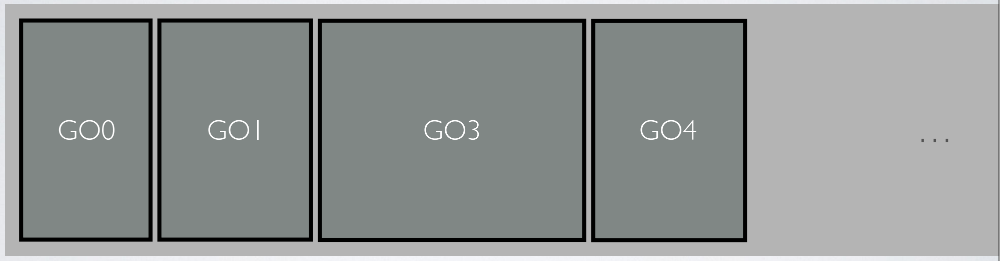


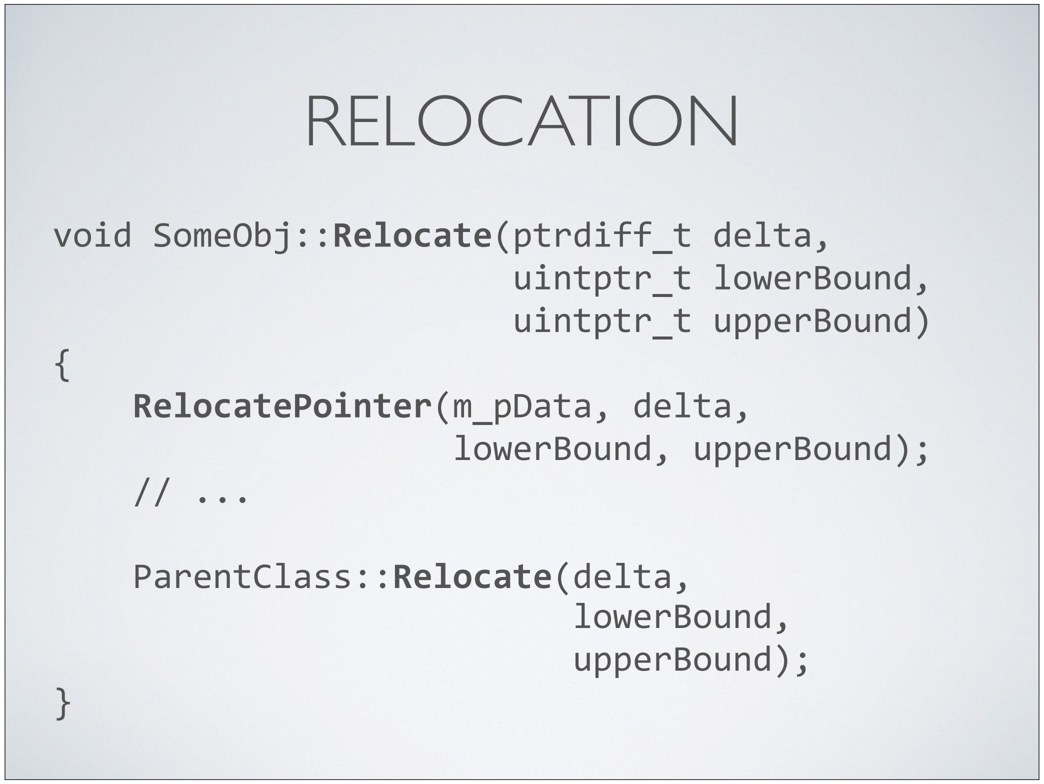
RELOCATION

RELOCATION



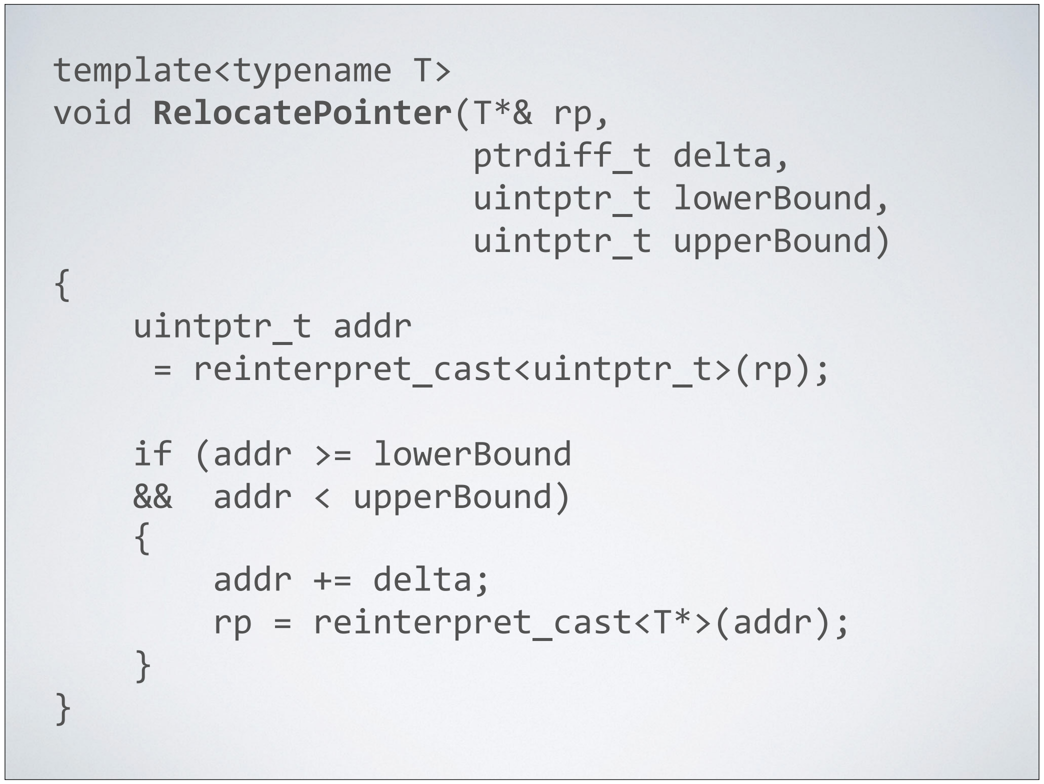
RELOCATION





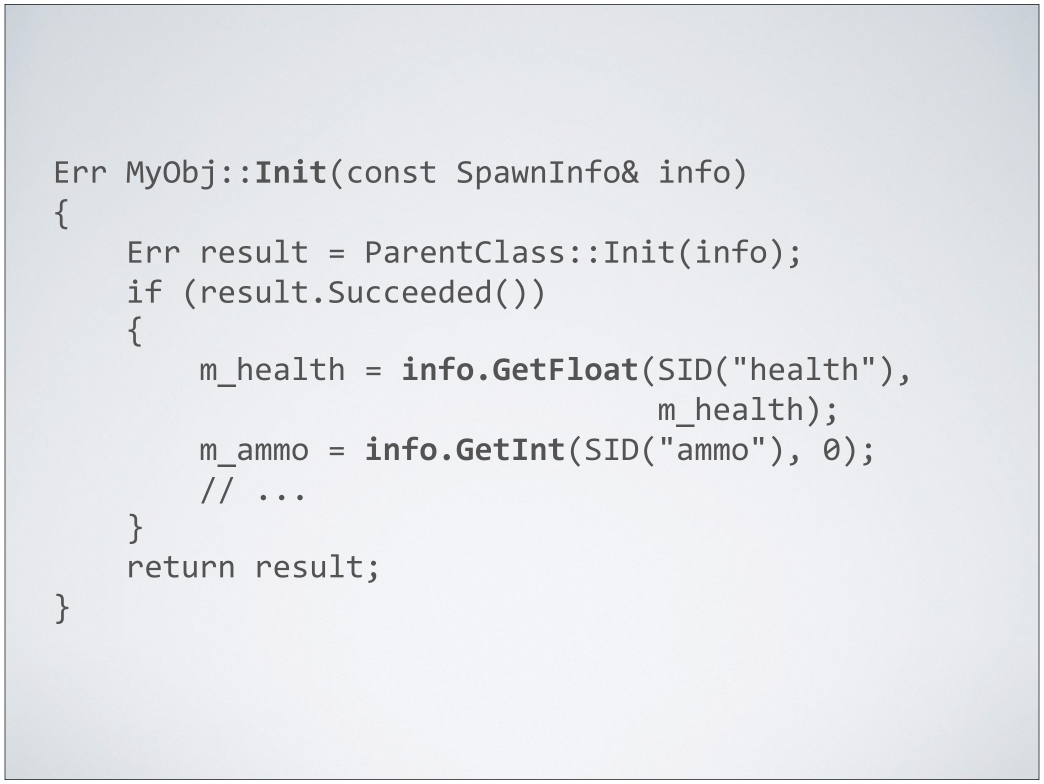
RELOCATION

```
void SomeObj::Relocate(ptrdiff_t delta,
                       uintptr_t lowerBound,
                       uintptr_t upperBound)
{
    RelocatePointer(m_pData, delta,
                    lowerBound, upperBound);
    // ...

    ParentClass::Relocate(delta,
                          lowerBound,
                          upperBound);
}
```


```
template<typename T>
void RelocatePointer(T*& rp,
                     ptrdiff_t delta,
                     uintptr_t lowerBound,
                     uintptr_t upperBound)
{
    uintptr_t addr
     = reinterpret_cast<uintptr_t>(rp);

    if (addr >= lowerBound
    &&  addr < upperBound)
    {
        addr += delta;
        rp = reinterpret_cast<T*>(addr);
    }
}
```

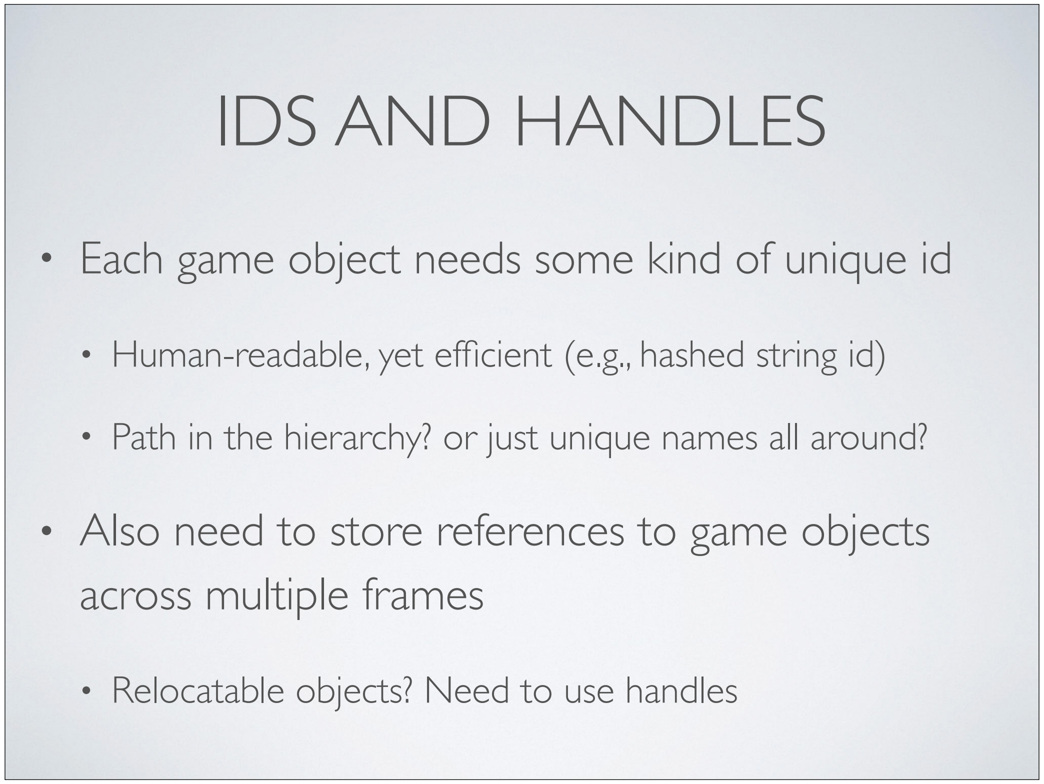
OBJECT INITIALIZATION

- Naughty Dog uses the following approach (YMMV):
- In game object's Init() function, it simply reads the **key-value store**
  - Flexible system, robust to tool-side changes
  - Init() is free to do any kind of initialization it wants
    - Create components, allocate per-instance data, etc.



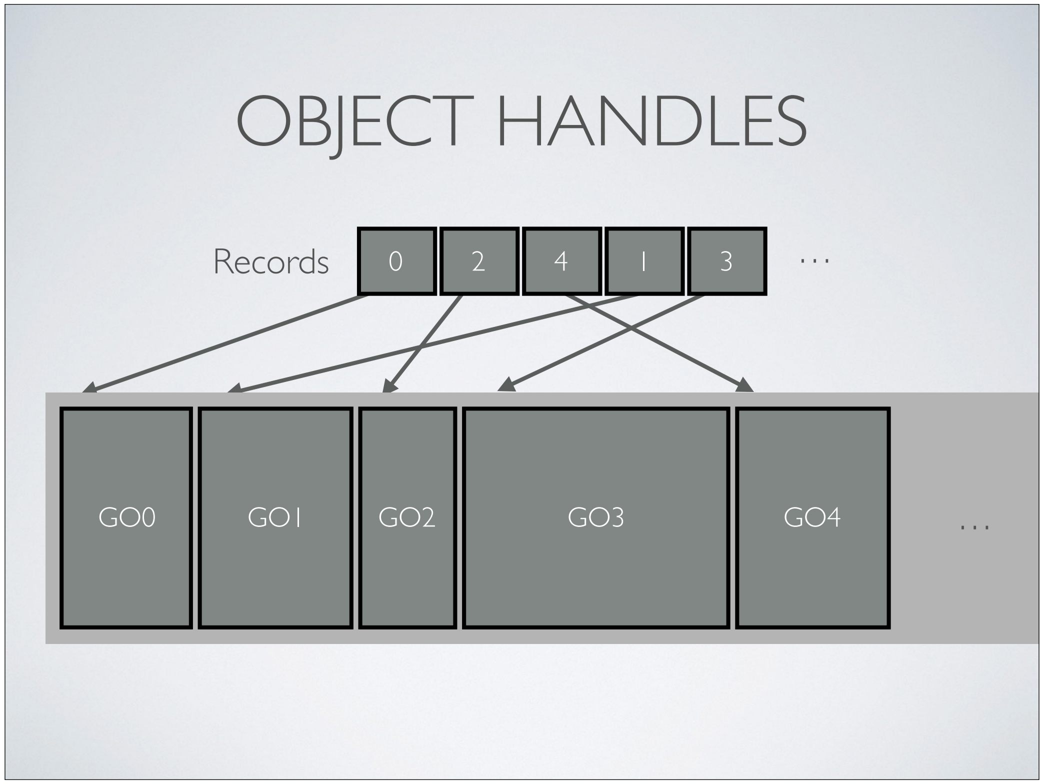
```
Err MyObj::Init(const SpawnInfo& info)
{
    Err result = ParentClass::Init(info);
    if (result.Succeeded())
    {
        m_health = info.GetFloat(SID("health"),
                                 m_health);
        m_ammo = info.GetInt(SID("ammo"), 0);
        // ...
    }
    return result;
}
```

GAME OBJECT REFERENCES


IDS AND HANDLES

- Each game object needs some kind of unique id
  - Human-readable, yet efficient (e.g., hashed string id)
  - Path in the hierarchy? or just unique names all around?
- Also need to store references to game objects across multiple frames
  - Relocatable objects? Need to use handles

we use SIDs as object unique ids (human-readable, yet efficient)



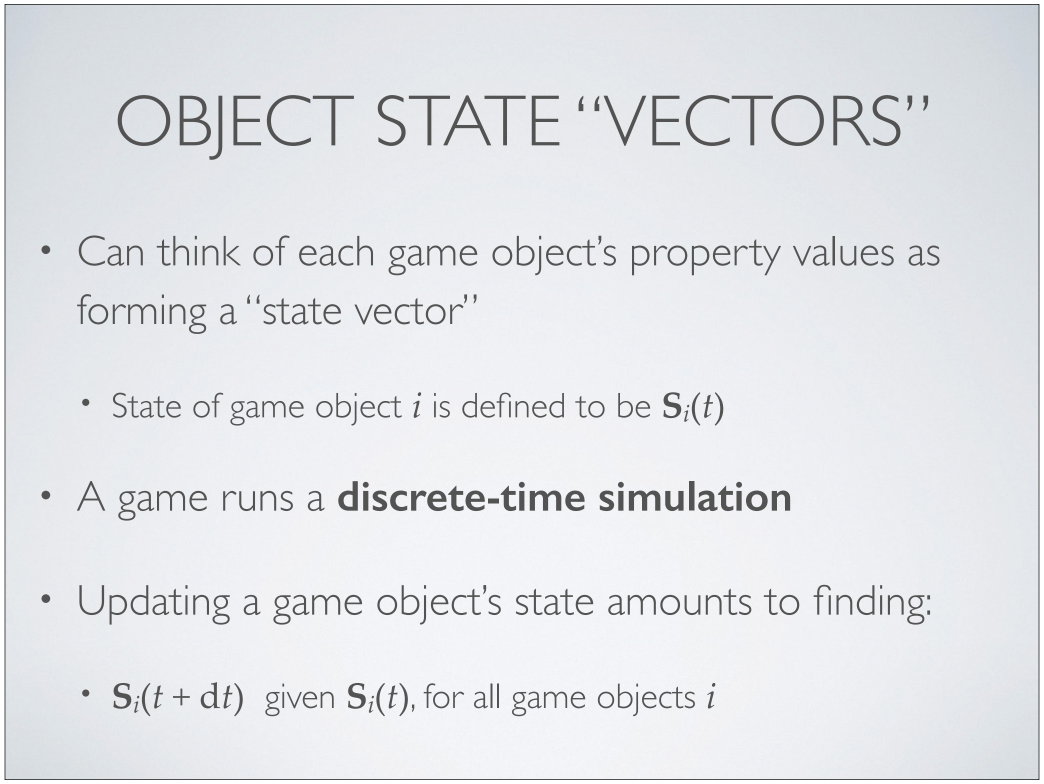
OBJECT HANDLES

record indices are fixed, never relocate

so can use a record index (or pointer to rec) as a “handle” to the GO

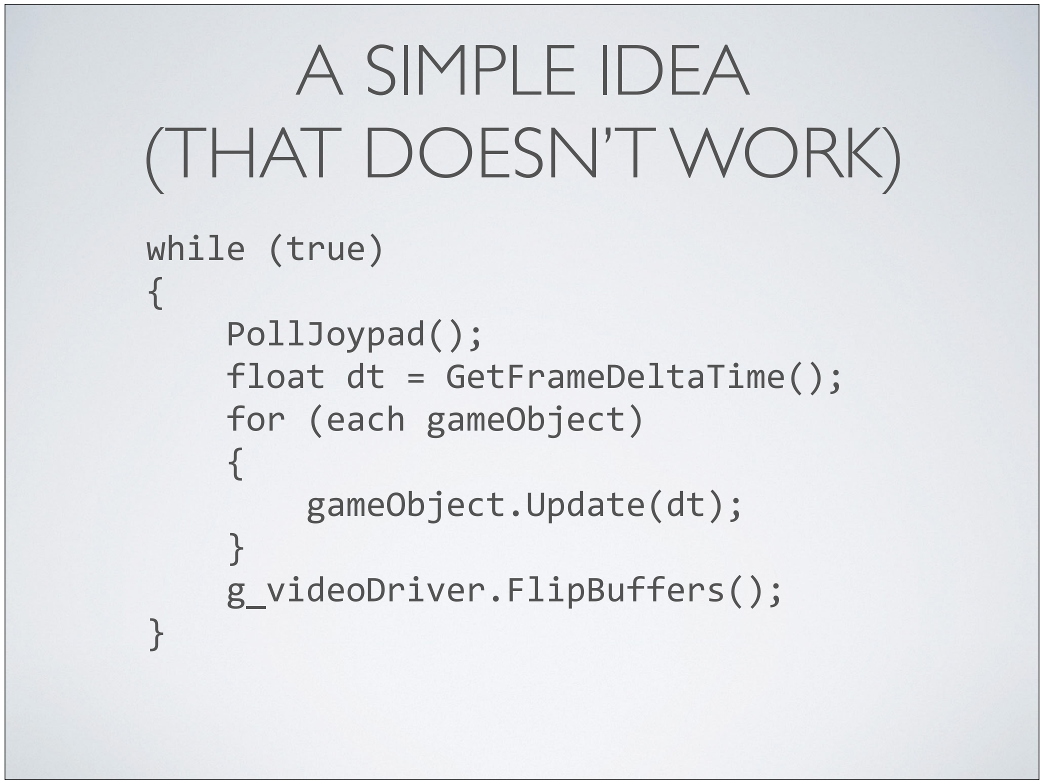
because GOs come and go, need to use the unique id of the GO as a verification that a handle hasn’t gone “stale” and been replaced with a new GO

UPDATING THE RUN-TIME OBJECT MODEL



OBJECT STATE "VECTORS"

- Can think of each game object's property values as forming a "state vector"
  - State of game object i is defined to be $\mathbf{S}_i(t)$
- A game runs a **discrete-time simulation**
- Updating a game object's state amounts to finding:
  - $\mathbf{S}_i(t + \mathrm{d}t)$ given $\mathbf{S}_i(t)$, for all game objects i



A SIMPLE IDEA (THAT DOESN'T WORK)

```
while (true)
{
    PollJoypad();
    float dt = GetFrameDeltaTime();
    for (each gameObject)
    {
        gameObject.Update(dt);
    }
    g_videoDriver.FlipBuffers();
}
```

```
virtual void Tank::Update(float dt)
{
    // Update the state of the tank itself.
    MoveTank(dt);
    DeflectTurret(dt);
    FireIfNecessary();

    // Now update low-level engine subsystems on
    // behalf of this tank. (NOT a good idea!)
    m_pAnimationComponent->Update(dt);
    m_pCollisionComponent->Update(dt);
    m_pPhysicsComponent->Update(dt);
    m_pAudioComponent->Update(dt);
    m_pRenderingComponent->draw();
}
```

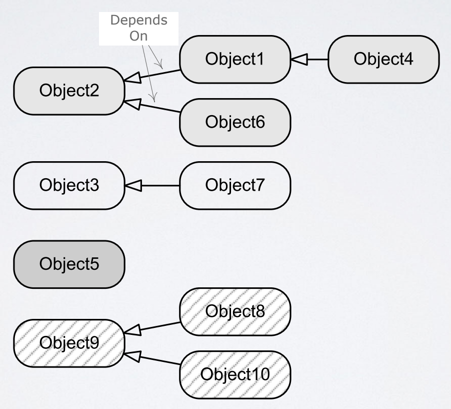
BATCHED UPDATES

- Most engine subsystems have tight performance constraints
- Much more efficient to do all updates of a certain variety at once as a "batch"
  - Code and data locality
    - Improves I-cache and D-cache performance
  - Minimizes duplicated calculations
  - Optimal data pipelining

BATCHED GAME LOOP

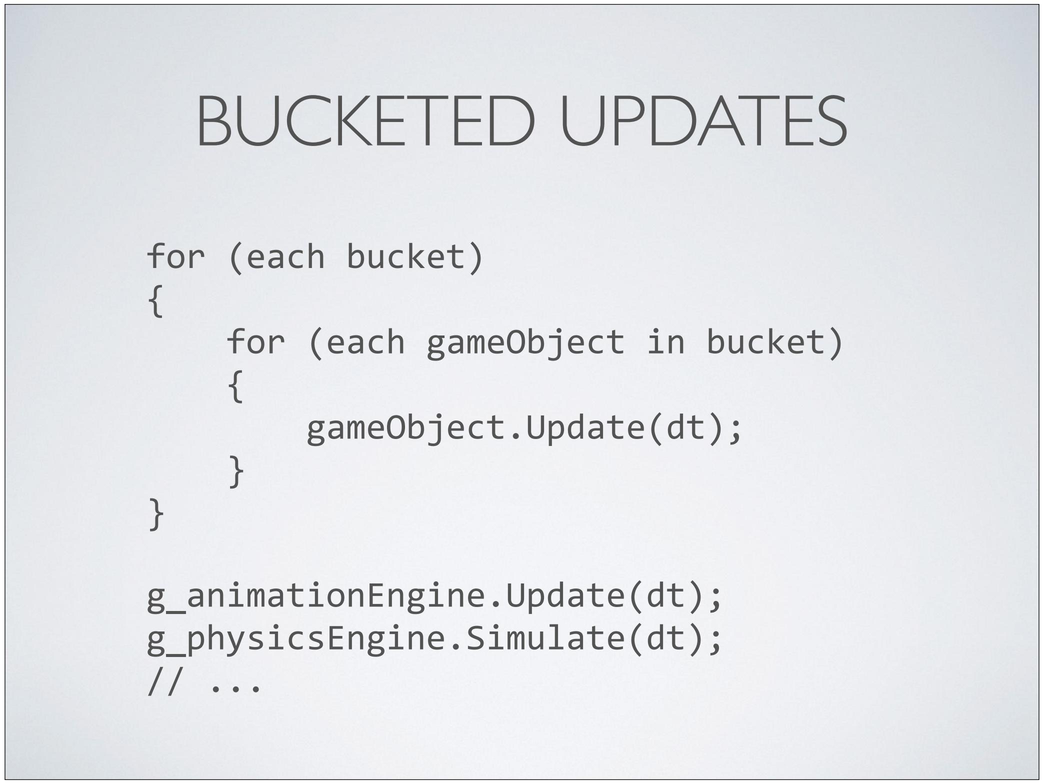
```
while (true)
{
    PollJoypad();
    float dt = GetFrameDeltaTime();
    for (each gameObject)
    {
        gameObject.Update(dt);
    }
    g_animationEngine.Update(dt);
    g_physicsEngine.Simulate(dt);
    g_collisionEngine.Run(dt);
    g_audioEngine.Update(dt);
    g_renderingEngine.RenderFrame();
    g_videoDriver.FlipBuffers();
 }
```

INTER-OBJECT DEPENDENCIES



another issue to consider is that objects depend on one another

can’t update Object1 until Object2 has been updated, and its new state vector is known

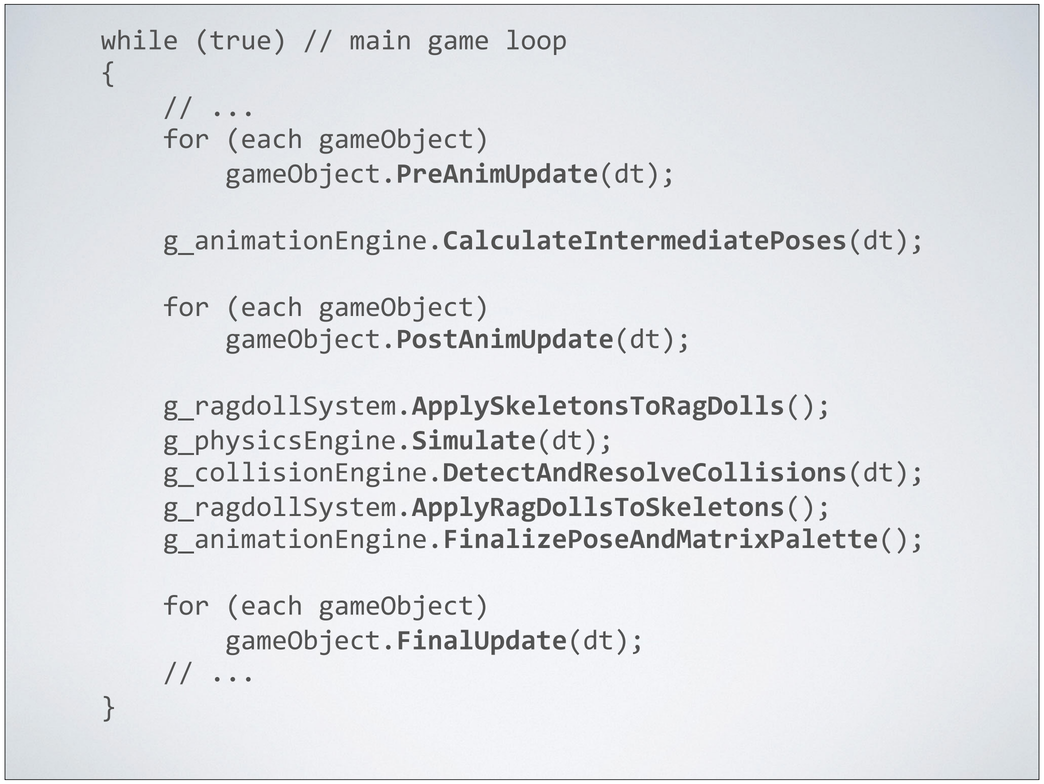


BUCKETED UPDATES

```
for (each bucket)
{
    for (each gameObject in bucket)
    {
        gameObject.Update(dt);
    }
}

g_animationEngine.Update(dt);
g_physicsEngine.Simulate(dt);
// ...
```

PHASED UPDATES

- Must also consider interactions between game objects and other engine subsystems
- Some subsystems may update in **phases**
  - e.g., **Animation** might operate like this:
    - Calculate intermediate poses
    - Apply poses to rag doll physics
    - Simulate physics/rag dolls
    - Apply rag doll final poses to skeletons
    - Post-process for procedural animation, IK, etc.
    - Generate final matrix palette

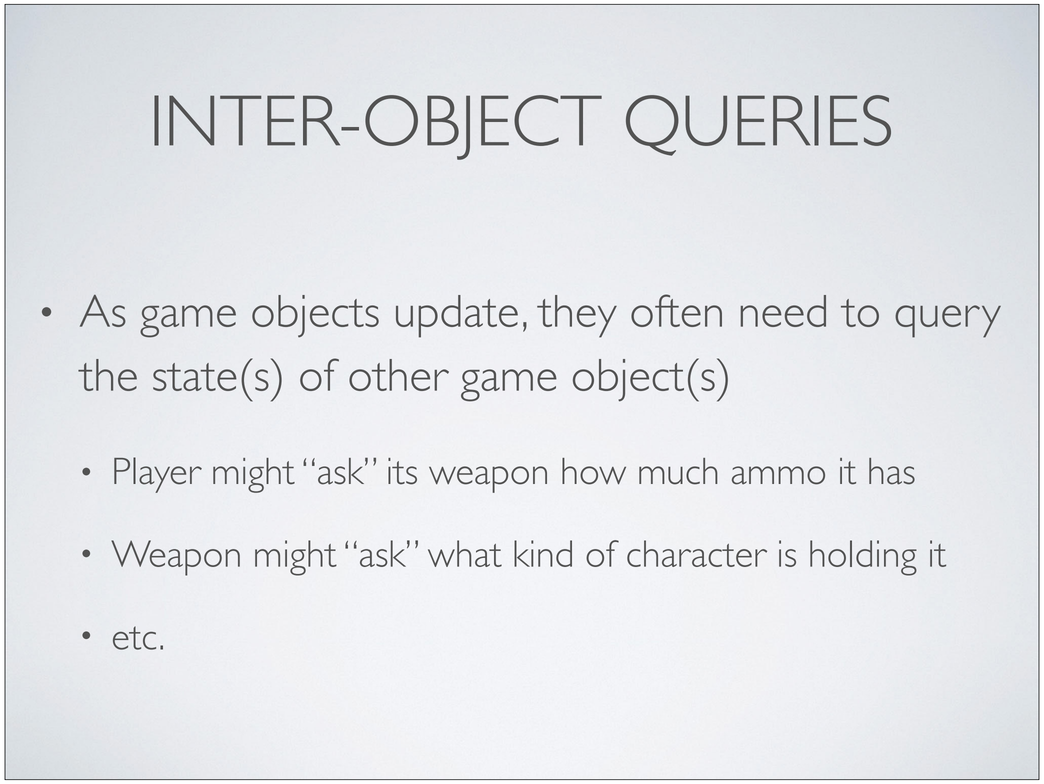


```
while (true) // main game loop
{
    // ...
    for (each gameObject)
        gameObject.PreAnimUpdate(dt);

    g_animationEngine.CalculateIntermediatePoses(dt);

    for (each gameObject)
        gameObject.PostAnimUpdate(dt);

    g_ragdollSystem.ApplySkeletonsToRagDolls();
    g_physicsEngine.Simulate(dt);
    g_collisionEngine.DetectAndResolveCollisions(dt);
    g_ragdollSystem.ApplyRagDollsToSkeletons();
    g_animationEngine.FinalizePoseAndMatrixPalette();

    for (each gameObject)
        gameObject.FinalUpdate(dt);
    // ...
}
```

each engine will differ somewhat

we can give our objects however many phases we need

also, combined with bucketed updates, some or all of the above may be done per-bucket

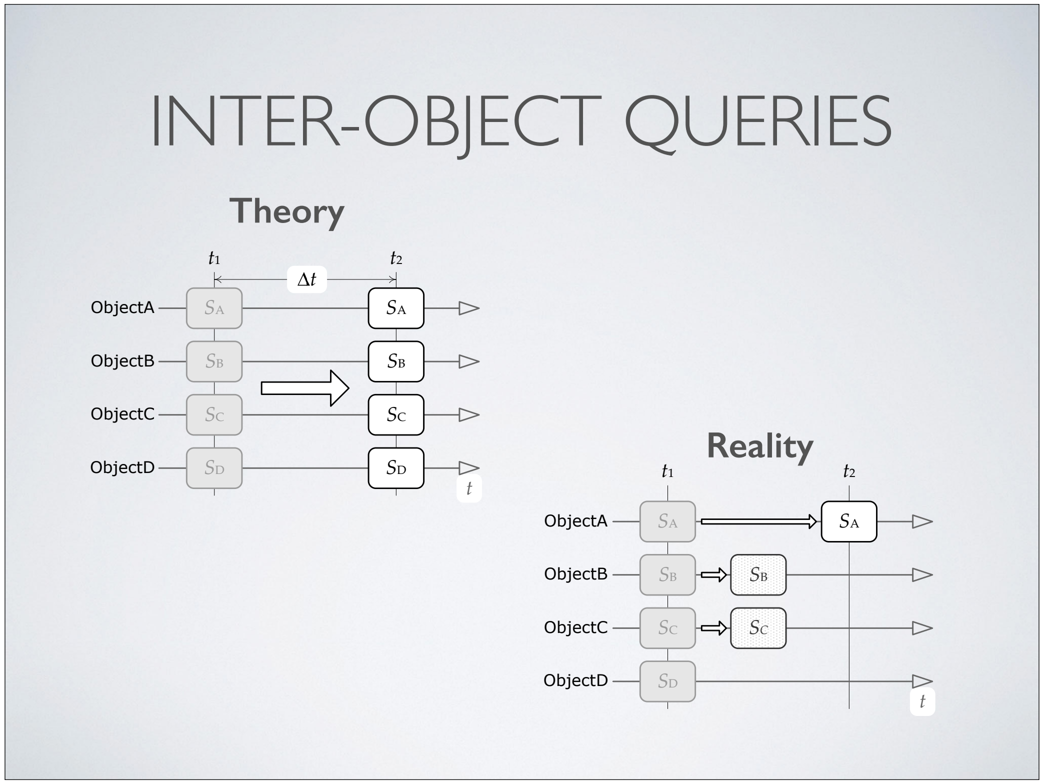


INTER-OBJECT QUERIES

- As game objects update, they often need to query the state(s) of other game object(s)
  - Player might "ask" its weapon how much ammo it has
  - Weapon might "ask" what kind of character is holding it
  - etc.

INTER-OBJECT QUERIES

- Having one game object query the state of another leads to all sorts of issues



INTER-OBJECT QUERIES

ideally updating happens simultaneously on all objects, totally independently

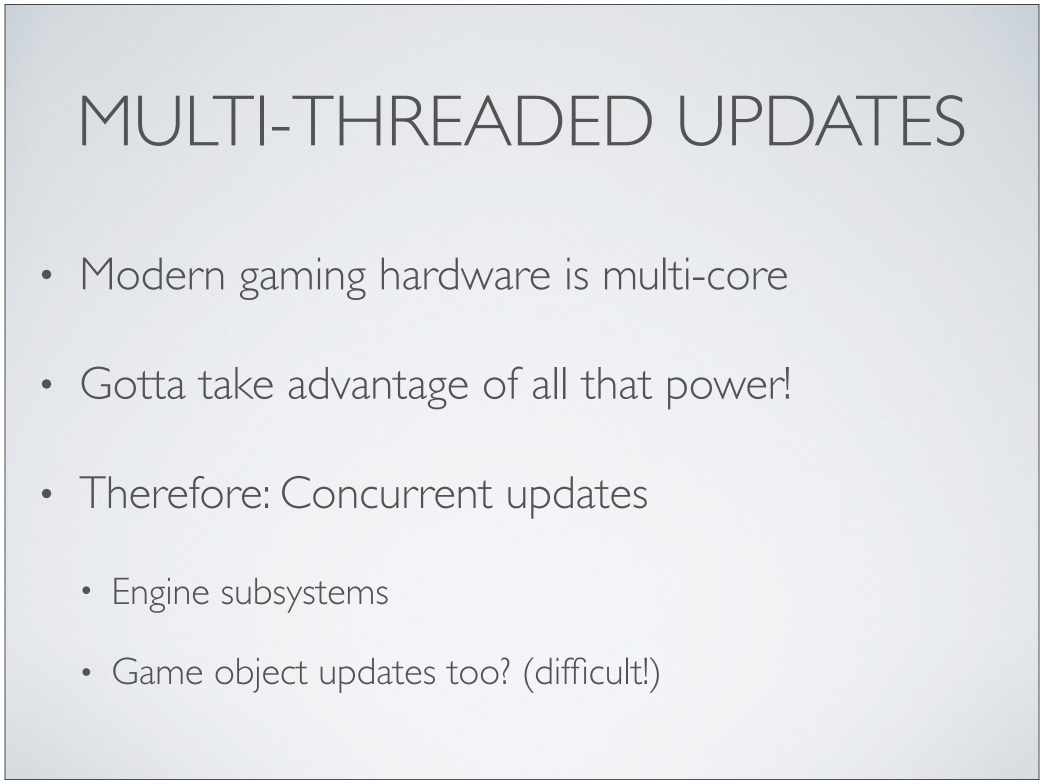
in reality, it happens piecemeal over the course of a single frame

INTER-OBJECT QUERIES

- The states of all game objects are consistent **before** and **after** the update loop, but they will be *inconsistent* **during** it.

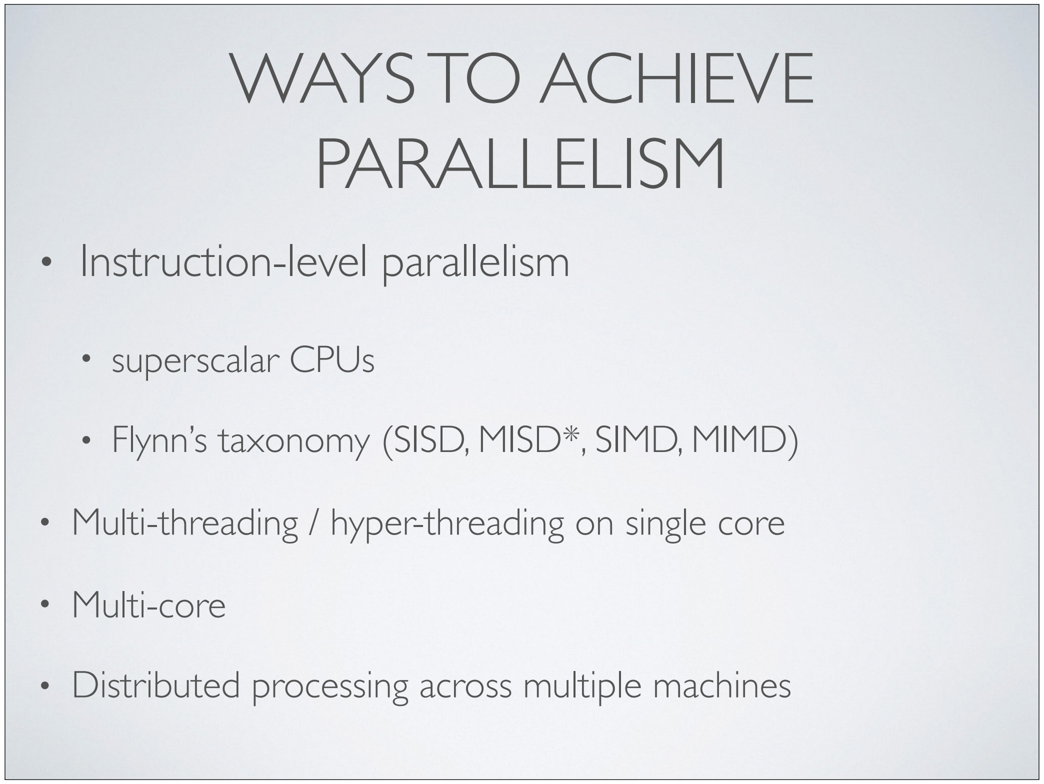
INTER-OBJECT QUERIES

- Some solutions to this problem:
  - Bucketed updates: Only query objects in *other* buckets
  - State cache: Keep a copy of last frame's (consistent) state
  - Just be careful out there: Deal with bugs if/as they happen!



MULTI-THREADED UPDATES

- Modern gaming hardware is multi-core
- Gotta take advantage of all that power!
- Therefore: Concurrent updates
  - Engine subsystems
  - Game object updates too? (difficult!)



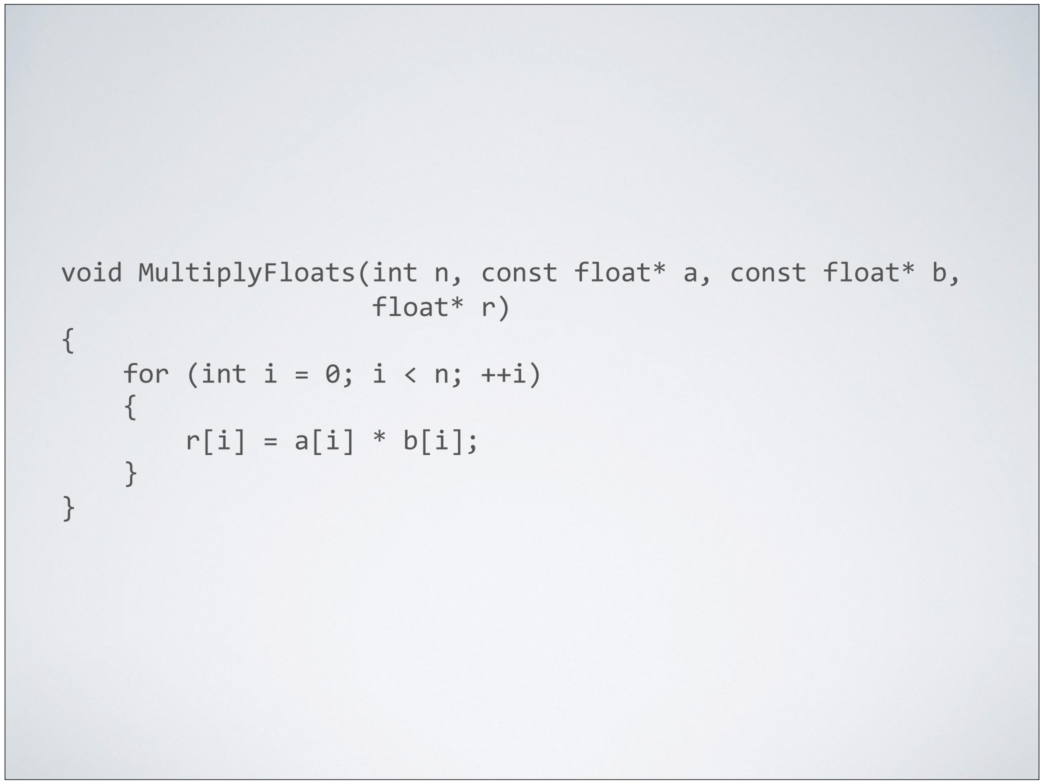
WAYS TO ACHIEVE PARALLELISM

- Instruction-level parallelism
  - superscalar CPUs
  - Flynn's taxonomy (SISD, MISD*, SIMD, MIMD)
- Multi-threading / hyper-threading on single core
- Multi-core
- Distributed processing across multiple machines

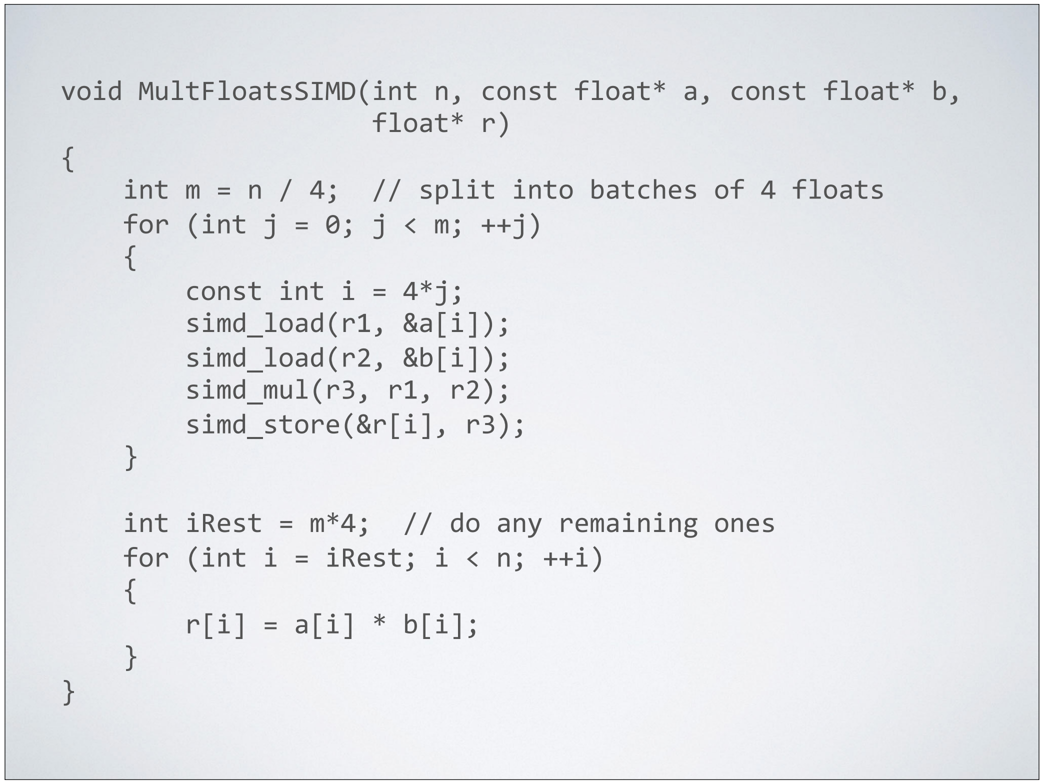
*MISD usually only used for fault tolerance, not relevant to games generally

SIMD

- SIMD vector processing available on most modern CPUs
- Can use to do 3D vector math (natural when your SIMD is 4-channel)
- Can also divide your work into parallel streams (4-channel or higher SIMD)

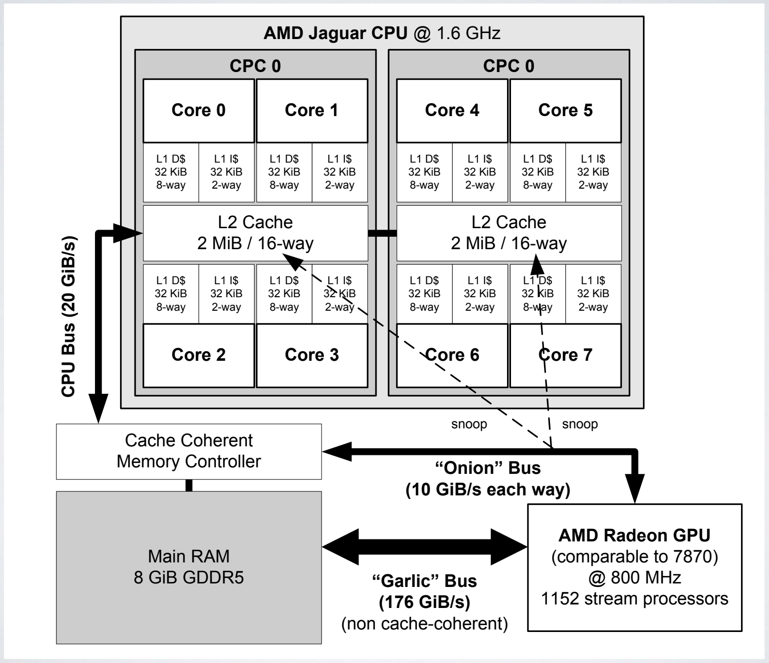


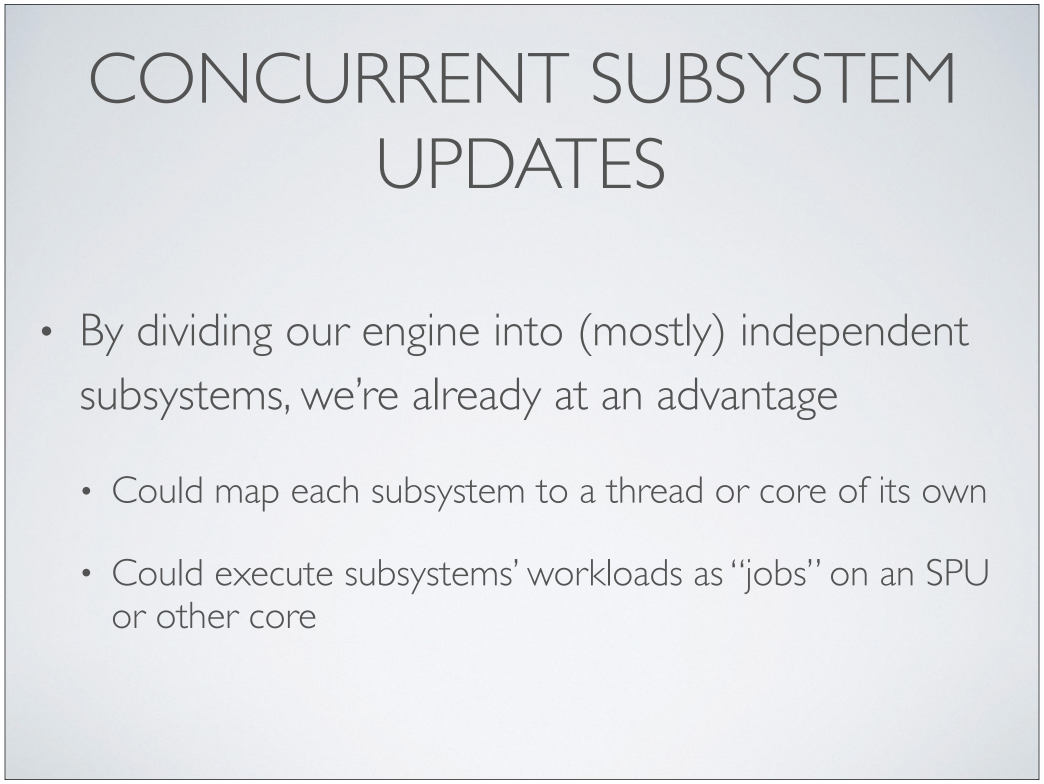
```
void MultiplyFloats(int n, const float* a, const float* b,
                    float* r)
{
    for (int i = 0; i < n; ++i)
    {
        r[i] = a[i] * b[i];
    }
}
```



```
void MultFloatsSIMD(int n, const float* a, const float* b,
                    float* r)
{
    int m = n / 4;  // split into batches of 4 floats
    for (int j = 0; j < m; ++j)
    {
        const int i = 4*j;
        simd_load(r1, &a[i]);
        simd_load(r2, &b[i]);
        simd_mul(r3, r1, r2);
        simd_store(&r[i], r3);
    }

    int iRest = m*4;  // do any remaining ones
    for (int i = iRest; i < n; ++i)
    {
        r[i] = a[i] * b[i];
    }
}
```

MULTI-CORE ARCHITECTURES





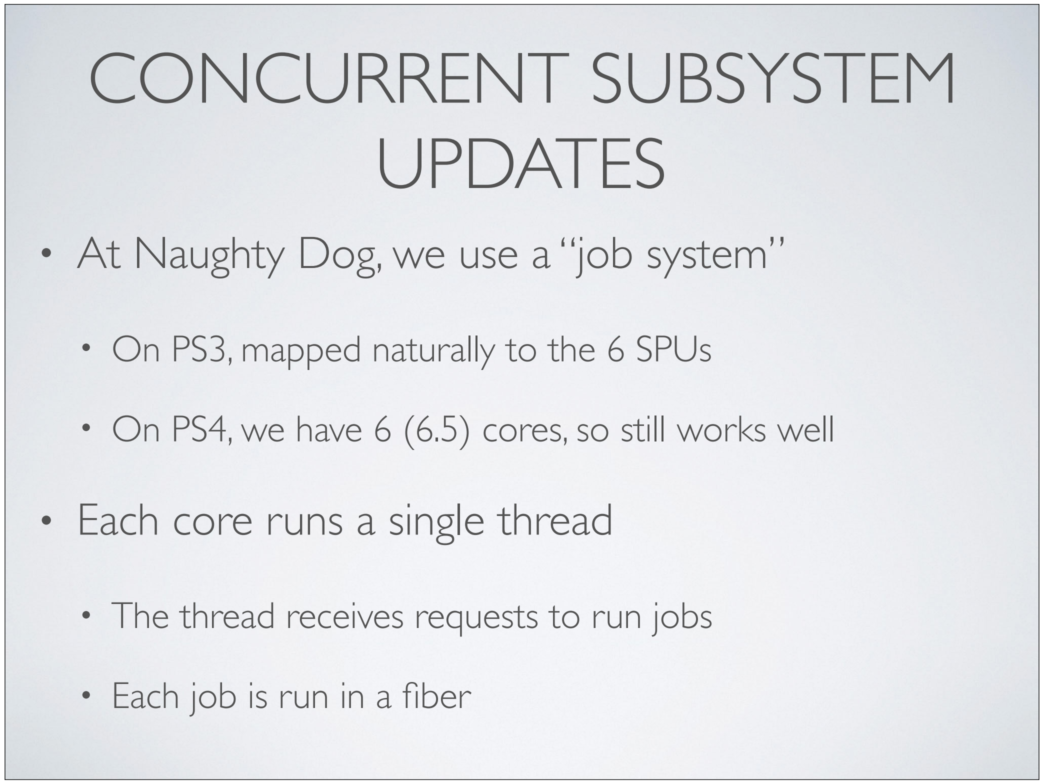
CONCURRENT SUBSYSTEM UPDATES

- By dividing our engine into (mostly) independent subsystems, we're already at an advantage
  - Could map each subsystem to a thread or core of its own
  - Could execute subsystems' workloads as "jobs" on an SPU or other core

This maps very naturally to subsystems like animation, path finding, ray casting
With some work, we were able to get Havok to fit into our system as well

Works well any time the subsystems are mostly independent
* well-defined input -> processing -> well-defined output

Not so easy to update game objects this way, b/c of large degree of inter-dependency



CONCURRENT SUBSYSTEM UPDATES

- At Naughty Dog, we use a “job system”
  - On PS3, mapped naturally to the 6 SPUs
  - On PS4, we have 6 (6.5) cores, so still works well
- Each core runs a single thread
  - The thread receives requests to run jobs
  - Each job is run in a fiber

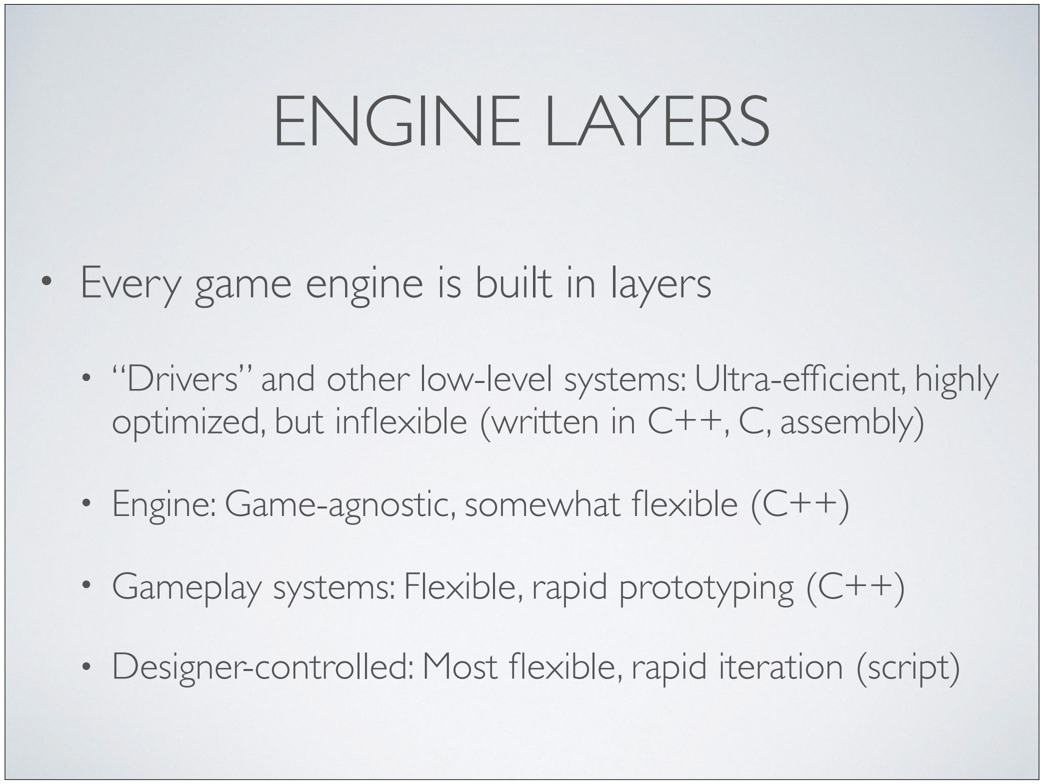
JOB SYSTEM SYNCHRONIZATION

- Counters
  - Each job increments the counter when it runs, decrements when done
  - Another "client" job can wait for the counter to reach zero
- Spin locks
  - Implemented via atomic operations provided by CPU
  - Exclusive locks; also multiple-reader, single writer

CONCURRENT GAME OBJECT UPDATING

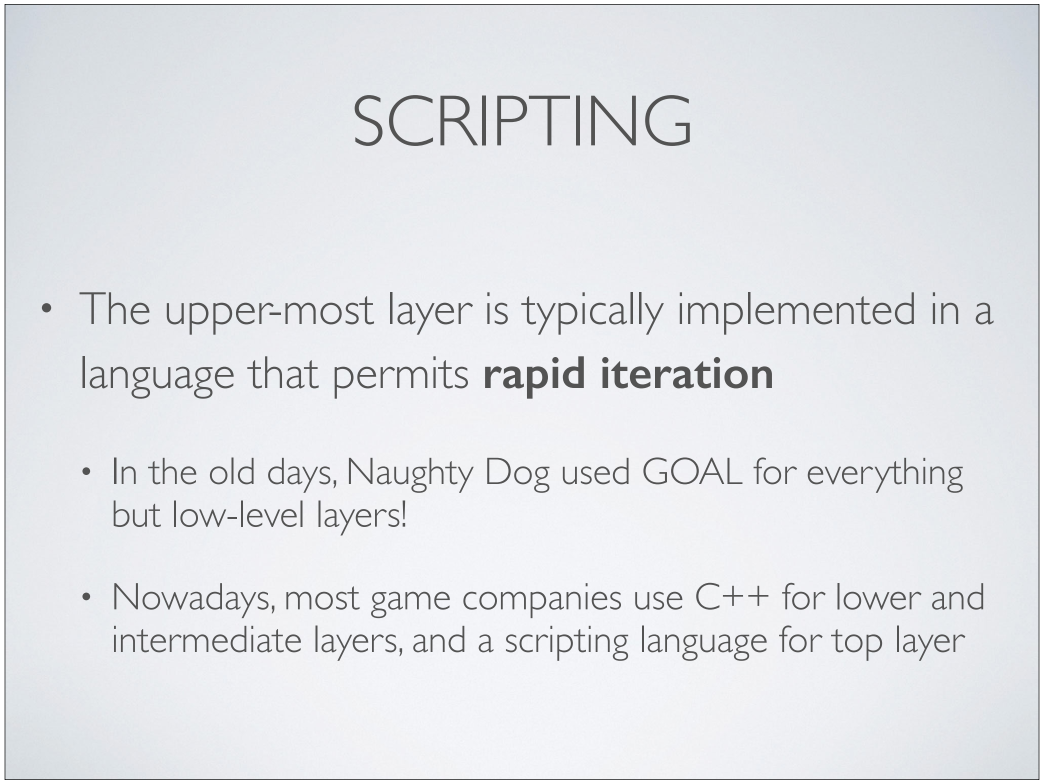
- Difficult to achieve because of high degree of inter-object dependencies and queries
- Some approaches that can work:
  - Locking (doesn't work very well in general)
  - Double-buffered game object state vectors
  - Snapshots (poor-man's double buffering)

SCRIPTING



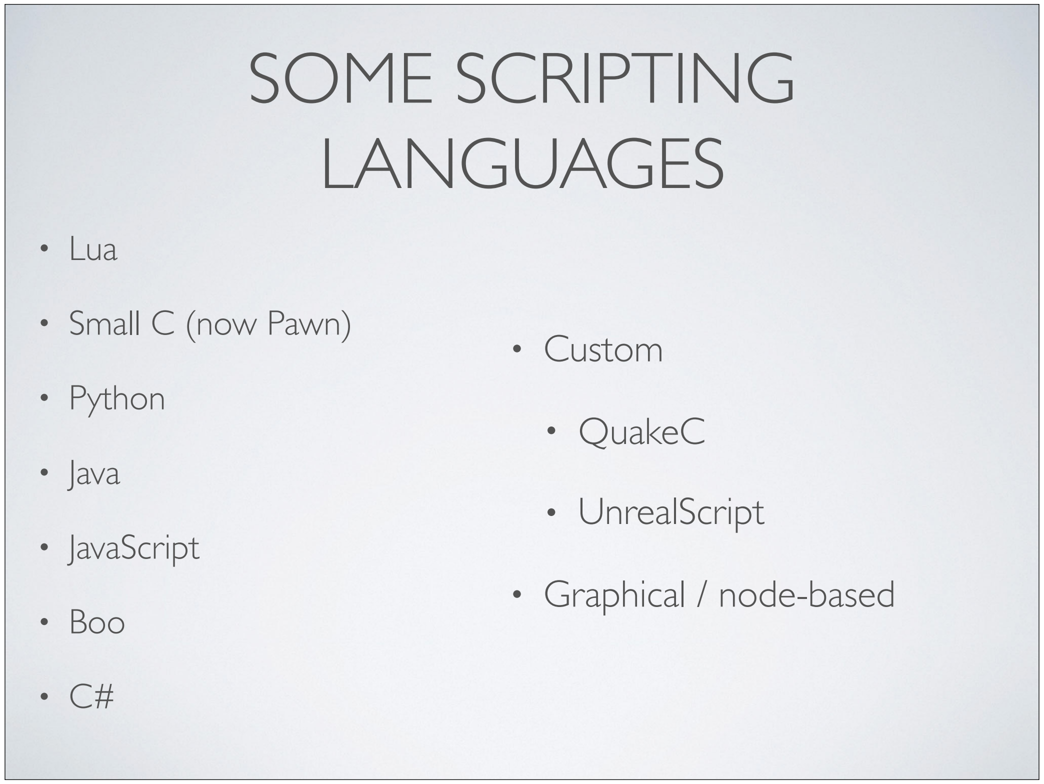
ENGINE LAYERS

- Every game engine is built in layers
  - "Drivers" and other low-level systems: Ultra-efficient, highly optimized, but inflexible (written in C++, C, assembly)
  - Engine: Game-agnostic, somewhat flexible (C++)
  - Gameplay systems: Flexible, rapid prototyping (C++)
  - Designer-controlled: Most flexible, rapid iteration (script)



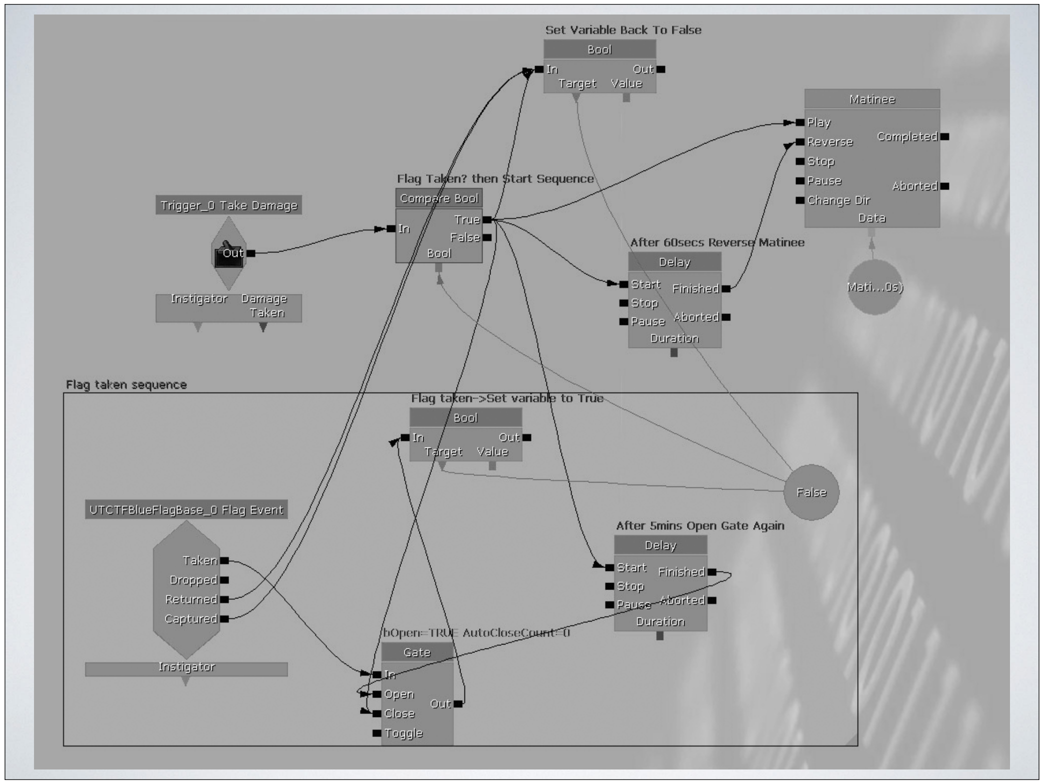
SCRIPTING

- The upper-most layer is typically implemented in a language that permits **rapid iteration**
  - In the old days, Naughty Dog used GOAL for everything but low-level layers!
  - Nowadays, most game companies use C++ for lower and intermediate layers, and a scripting language for top layer



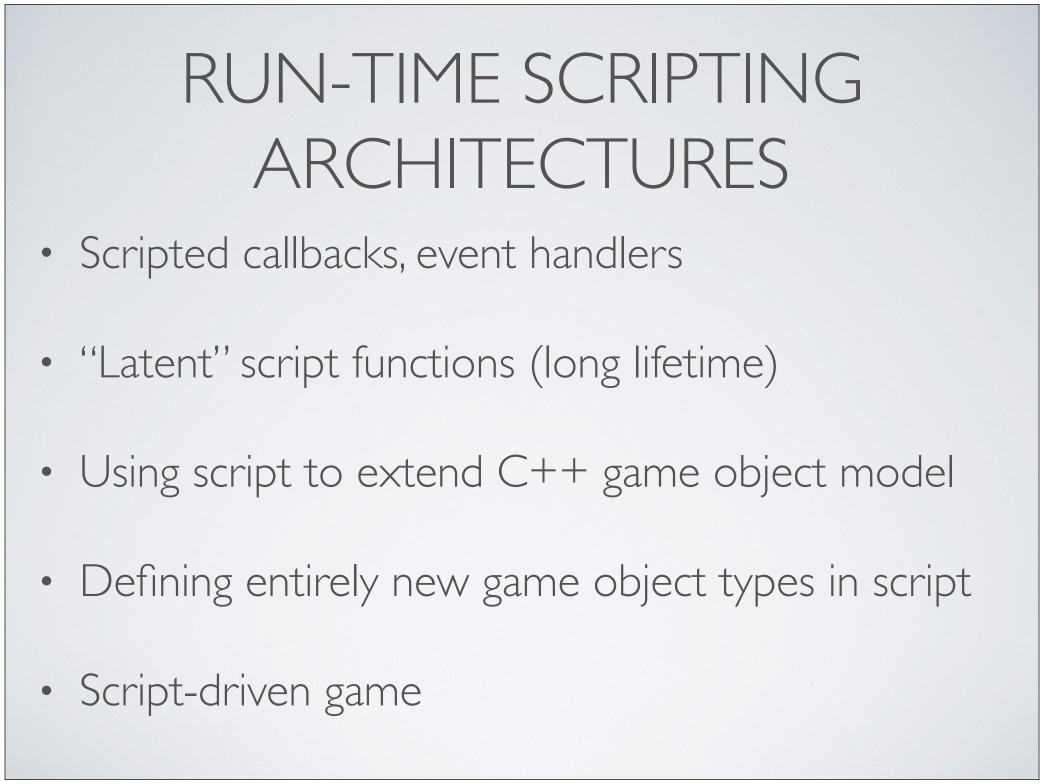
SOME SCRIPTING LANGUAGES

- Lua
- Small C (now Pawn)
- Python
- Java
- JavaScript
- Boo
- C#
- Custom
  - QuakeC
  - UnrealScript
- Graphical / node-based


Set Variable Back To False
Bool
In
Out
Target
Value
Matinee
Play
Reverse
Stop
Pause
Change Dir
Completed
Aborted
Data
Mati...0s)
Flag Taken? then Start Sequence
Compare Bool
In
True
False
Bool
Trigger_0 Take Damage
Out
Instigator
Damage Taken
After 60secs Reverse Matinee
Delay
Start
Stop
Pause
Finished
Aborted
Duration
Flag taken sequence
Flag taken->Set variable to True
Bool
In
Out
Target
Value
False
UTCTFBlueFlagBase_0 Flag Event
Taken
Dropped
Returned
Captured
Instigator
After 5mins Open Gate Again
Delay
Start
Stop
Pause
Finished
Aborted
Duration
bOpen=TRUE AutoCloseCount=0
Gate
In
Open
Close
Toggle
Out

TYPES OF SCRIPTING

- Data definition
- Run-time script



RUN-TIME SCRIPTING ARCHITECTURES

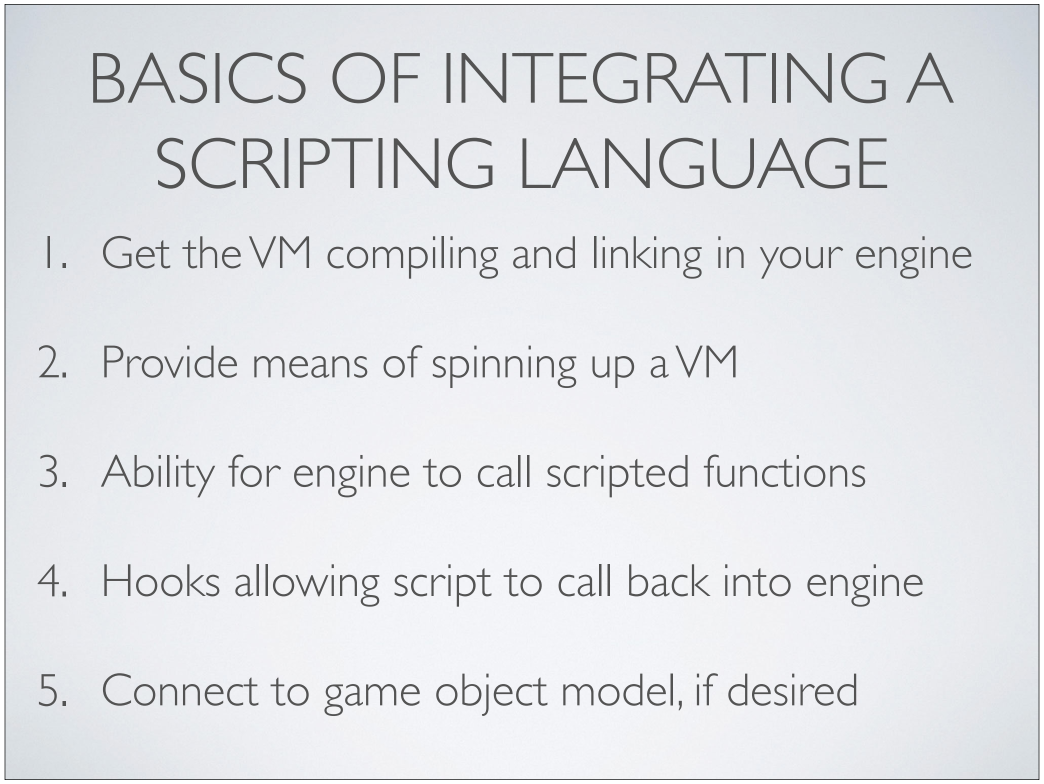
- Scripted callbacks, event handlers
- "Latent" script functions (long lifetime)
- Using script to extend C++ game object model
- Defining entirely new game object types in script
- Script-driven game

At its simplest, script "snippets" can be called by engine in response to events… gives the script a chance to react, call functions in the engine, then exit

"Latent" script functions are functions that start executing, then go to "sleep" for a time, and are woken back up by the engine in response to some condition being met

Can also extend the game object model itself using script / define entirely new types of GO!

Entire high-level game logic could be run by script — the engine could just respond to script commands, but never do anything itself



BASICS OF INTEGRATING A SCRIPTING LANGUAGE

1. Get the VM compiling and linking in your engine
2. Provide means of spinning up a VM
3. Ability for engine to call scripted functions
4. Hooks allowing script to call back into engine
5. Connect to game object model, if desired

#5 can be a big job

* how does one reference a game object within script? guid? handle? by name?
* how do you associate a script with a GO?
* what IS a script, anyway? single function? collection of functions / object? FSM?
* can a script-writer actually extend the object model (create a purely scripted subclass)?