Gameplay Systems & Runtime Gameplay

Mads Johansen Lassen

Agenda

- Part 1: Anatomy of Gameplay Systems
- Part 2: Game World Editor
- Part 3: Runtime Gameplay Foundation Systems
- Part 4: Game World Data Handling
- Exercise: Making a Component based Game Object structure

Part 1:
Anatomy of Gameplay Systems

Anatomy of a Game World

- Game Mechanics
- Backgrounds
- Static Elements
- Dynamic Elements

Game Mechanics

- The set of rules that govern the interactions between various entities in the game.
- Objectives of the Player
- Criteria for Success and Failure
- Player Abilities
- Types of Game Objects in the Game World
- Overall Gameflow

Backgrounds

- Used to give the illusion of a coherent world.
- Camera backgrounds
  - Solid Color
  - Static / Continous / Parallaxing Images (2D)
  - Skyboxes (3D)

Static Elements

- Things that don’t move in the Game world.
- Terrain, Trees, Buildings, Bridges, Roads, etc.
- Anything that doesn’t actively interact with gameplay.

Static Geometry

- Usually defined in tools like Blender or Maya
- Can be built out of Instanced Geometry
  - Small pieces that get combined in random patterns (eg. to create variety in a big wall)

World Chunks

- Large playable virtual worlds need to be divided somehow.
- World chunks (aka. levels, maps, stages or areas) are discrete playable regions
- Originally only one level could be loaded at a time, levels gave variety
- Chunks are good for memory management

Highlevel Game Flow

- The Games overall flow (eg. level1 -> level2 etc.)
- Defines a sequence, tree or graph of objectives
- Objectives are tasks, stages, levels or waves
- In old games there was a one-to-one between chunks and objectives (also why both are know as levels)

Dynamic Elements

- Things that move, are interact-able or player controlled.
- Characters, vehicles, weapons, items, particle emitters, baked navigation data, dynamic lights, etc.

Game Objects

- General term for any dynamic object in the Game World
- Static objects can also be Game Objects
- A collection of attributes and behaviours
- Many instances of the a specific Game Object type can live in a Game World

Runtime vs Tool-Side design

- Tool-Side object model is what the Editor uses, and Runtime is what the Game Engine uses
- The Runtime implementation of a Game Object, and the Tool-Side implementation don't need to be the same class
- Depends largely on game engine, the editor and the gameplay scripting language of choice
- Tool-Side objects might be nothing more than an id and some lookup tables at Runtime.

Data-Driven Gameplay

- Programmers are the most expensive resource on a team
- By using Data-Driven Gameplay, many tasks can be moved from Programmers to Designers and Artists
- Data-Driven Gameplay, means gameplay is defined and tweaked mostly using properties on Game Objects
- Be careful that the time spent creating a tool, is less than the time it saves in the long run

Part 2:

Game World Editor

Typical Features

- Game World Visualisation
  - Navigation
  - Selection of Game Objects
  - Layers
- Handles for placement and alignment
- Game Object Property View

Advanced features

- Level / Chunk / Scene Creation & Management
- Saving and Loading of Chunks / Levels / Scenes
- Special Object Types (Lights, Particle Emitters, Trigger Regions, Sound Sources, Splines, NavMesh)
- Rapid Iteration (fast switching to game mode)
- Asset Management Tools

Will you need to make an Editor?

- Alternatives for 3D or 2.5D
  - GtkRadiant (Quake editor family)
  - Blender3D (exports to COLLADA)
- Alternatives for 2D
  - DAME (Tile based)
  - Tile Map Editor (Tile based)
  - Overlap 2D
  - OGMO Editor

[BREAK]
10 MINS

Part 3:

Runtime Gameplay Foundation Systems

Gameplay Foundation System

- The base upon which all gameplay is built
- Needs to use the underlying Game Engine, to create the actual game
- No clearly defined boundary between game and engine

Typical Gameplay Systems

- Runtime Game Object Model
- Real-time Object Model Updating
- Messaging and Event Handling
- Scripting
- Objectives and gameflow management
- Level management and streaming

Runtime Object Model 1/2

- Spawning and destroying game objects
- Linkage to low-level engine systems (eg. render)
- Real-time simulation of object behaviours
- Ability to define new game objects
- Unique object ids

Runtime Object Model 2/2

- Game object queries (finding other objects)
- Game object references
- Finite State Machine support
- Network Replication
- Saving, Loading and Game Object Persistence

Runtime Object Model Architectures

- Object-centric style
  - Each game object has a runtime class instance, with a set of attributes and behaviours
- Property-centric style
  - Each Tool-Side game objet is represented by an id, and properties are distributed across many data tables, behaviours are implicitly defined by the collection of properties that comprises an object

Object-Centric Architecture

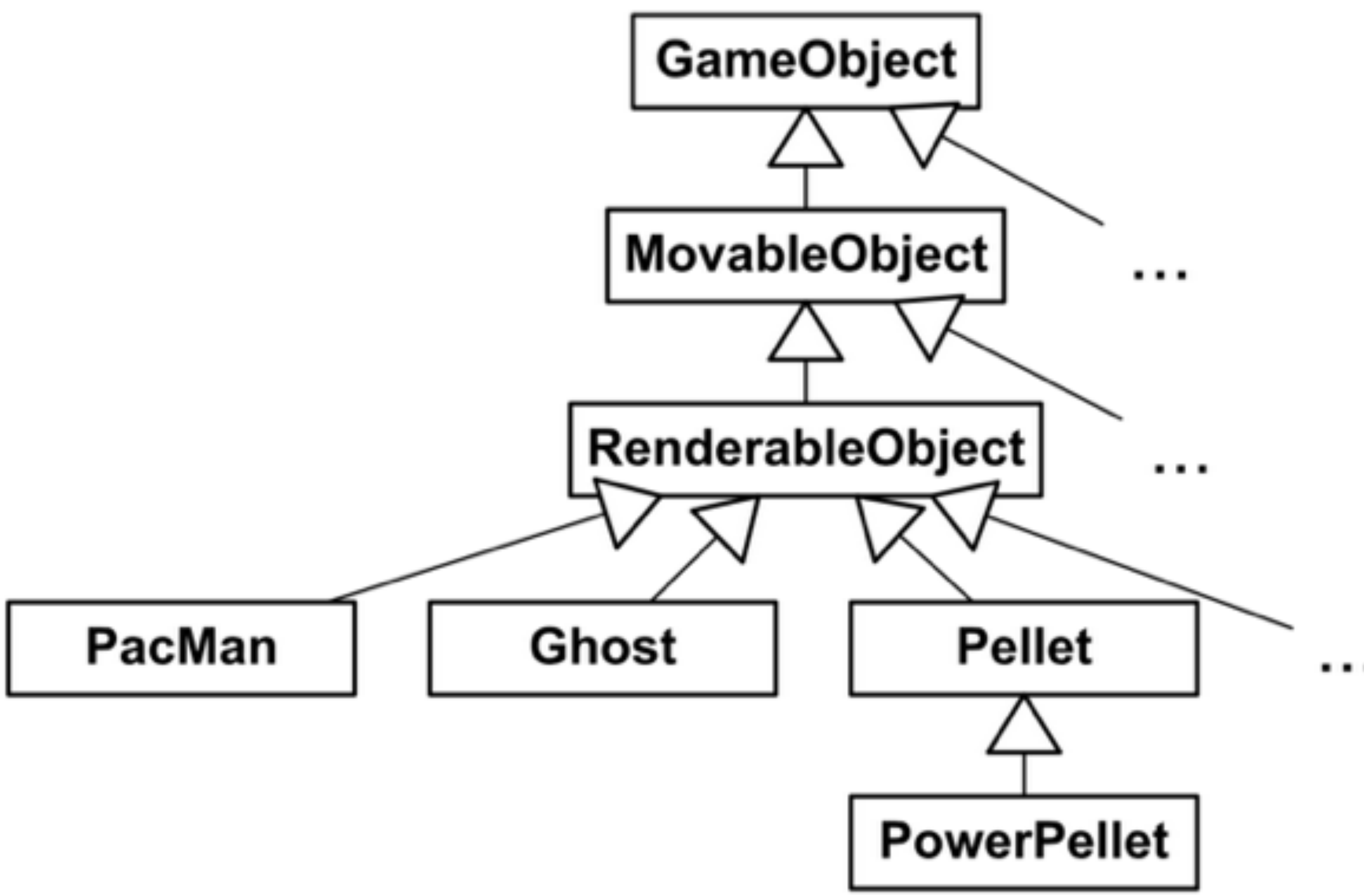
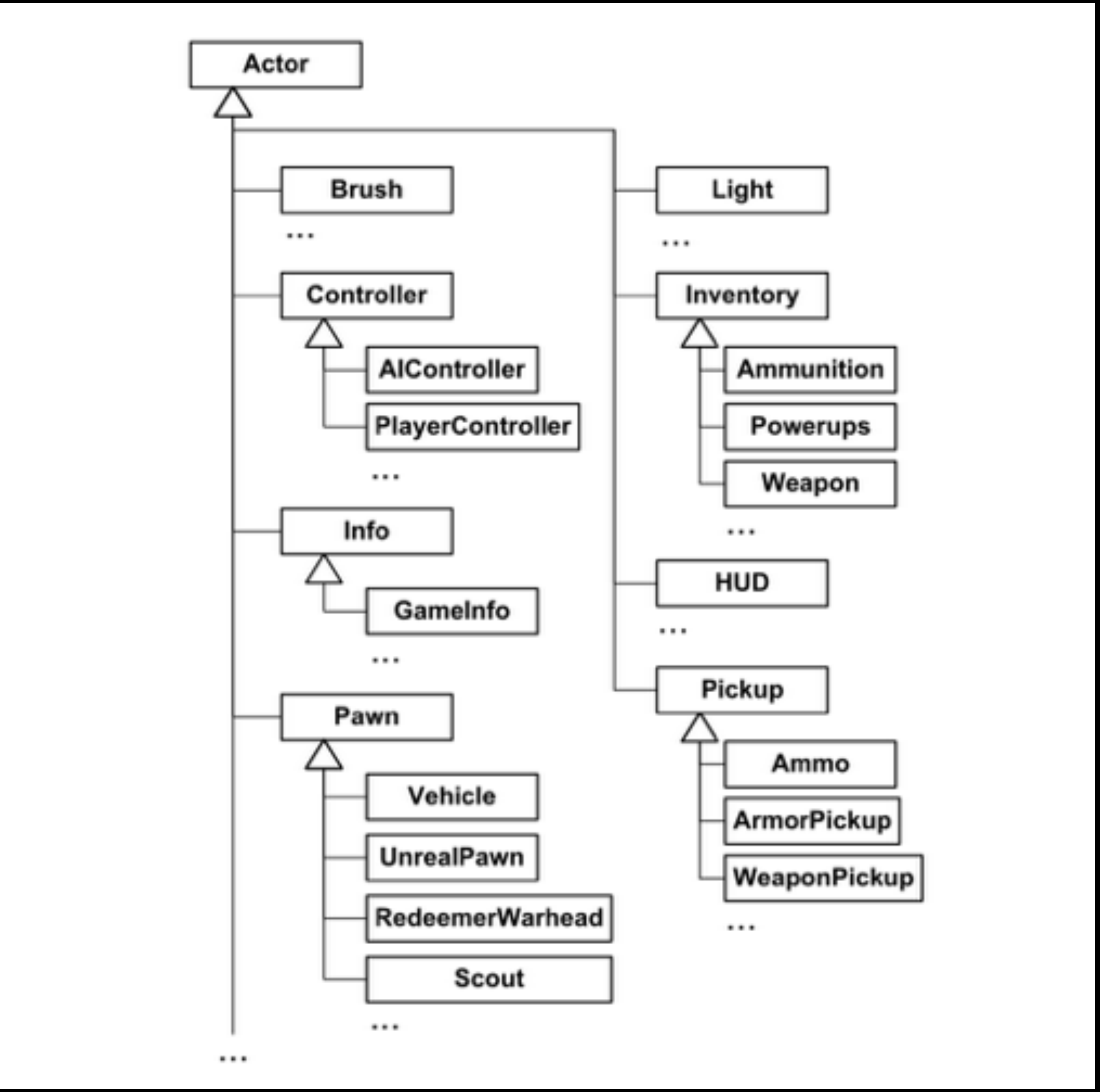


Figure 15.2. A hypothetical class hierarchy for the game *Pac-Man*.

Monolithic Class Hierarchy

- Begins small, but can easily explode
- Problems can arise because it is difficult to understand the consequences of changing base classes
- Unforeseen Multiple Inheritance Objects become hacks



Multiple Inheritance Issue

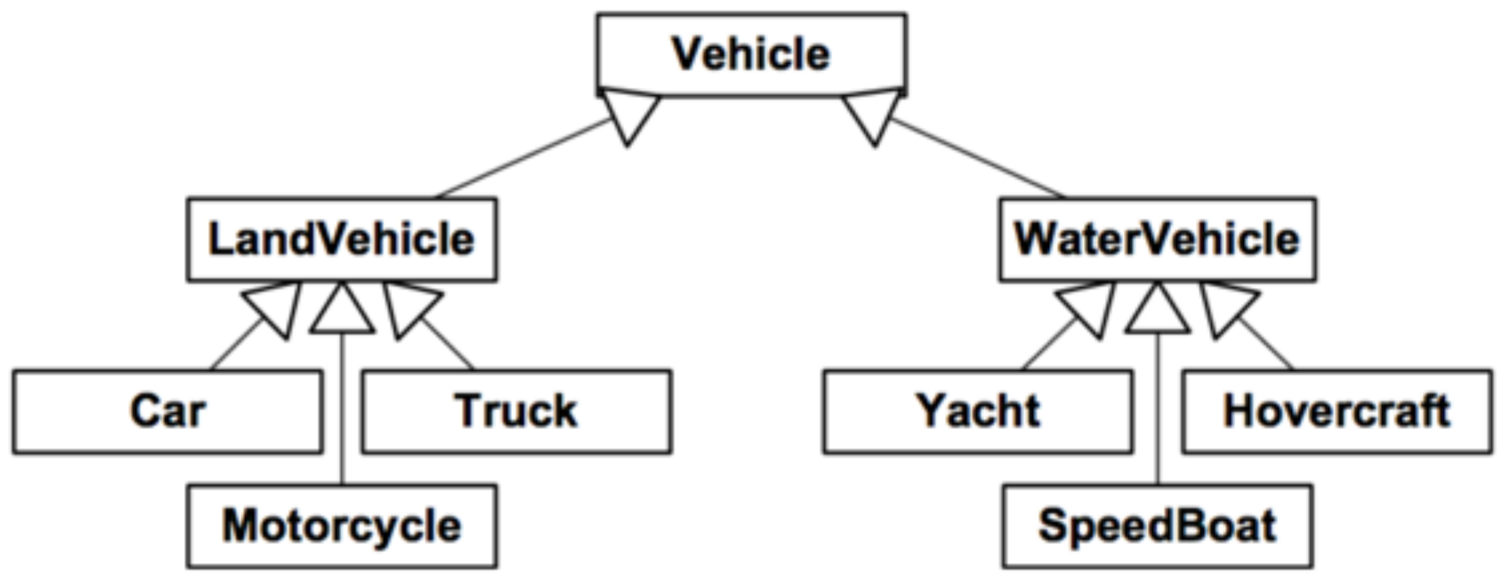


Figure 15.4. A seemingly logical class hierarchy describing various kinds of vehicles.

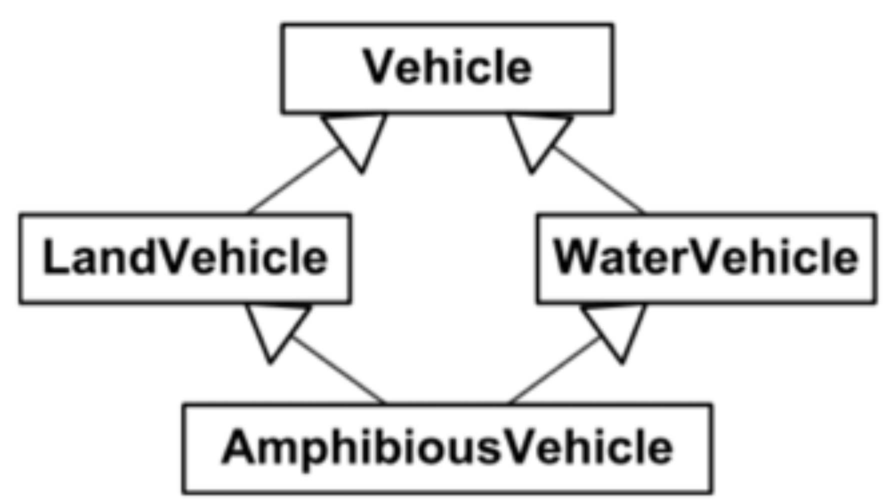


Figure 15.5. A diamond-shaped class hierarchy for amphibious vehicles.

Mix-In classes

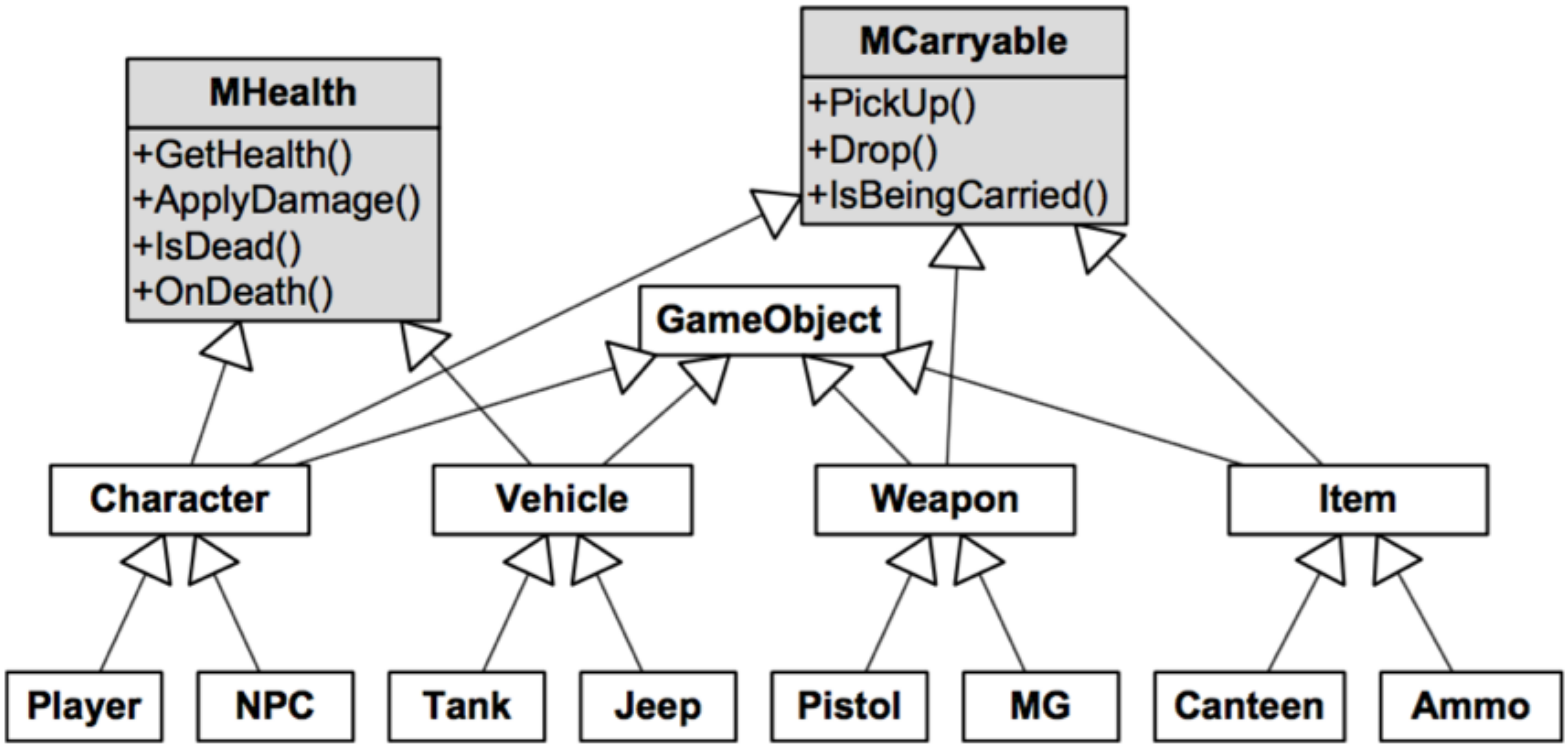


Figure 15.6. A class hierarchy with mix-in classes. The MHealth mix-in class adds the notion of health and the ability to be killed to any class that inherits it. The MCarryable mix-in class allows an object that inherits it to be carried by a Character.

The Bubble Up effect

- When building the class hierarchy, two otherwise unrelated object types, might need the same code
- The code gets moved to a shared baseclass
- After a while, the root object know about everything

Using Composition To Simplify

- Converting Is-A to Has-A
  - is-A checks inheritance
  - has-A checks components

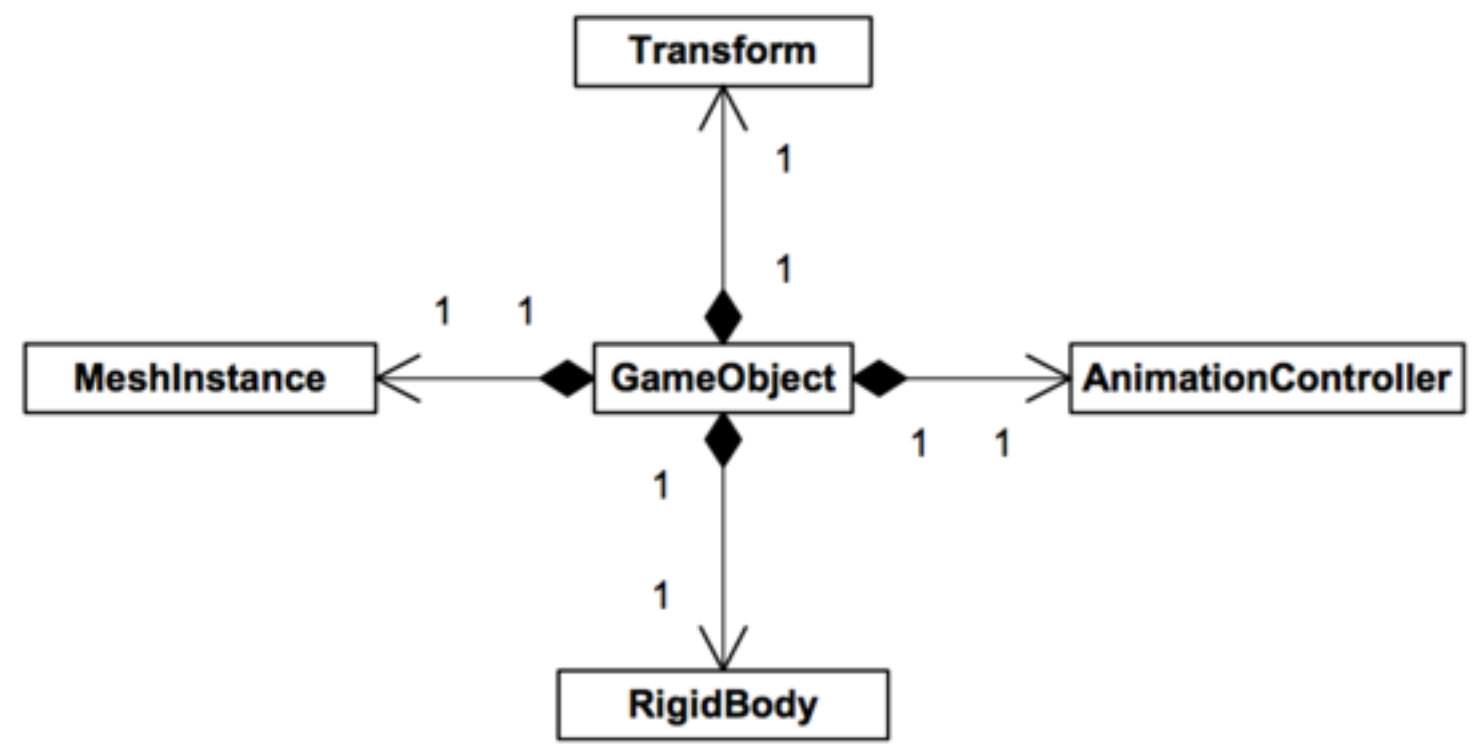


Figure 15.8. Our hypothetical game object class hierarchy, refactored to favor class composition over inheritance.

Component Creation and Ownership

- Either you create Game Object which is a central “hub” that knows about standard components
  - And then Inherit + extend in Child classes (great example on p. 883 - 885, in the book)
- Or you make a generic Component class that the Game Object then holds a list of

Generic Components

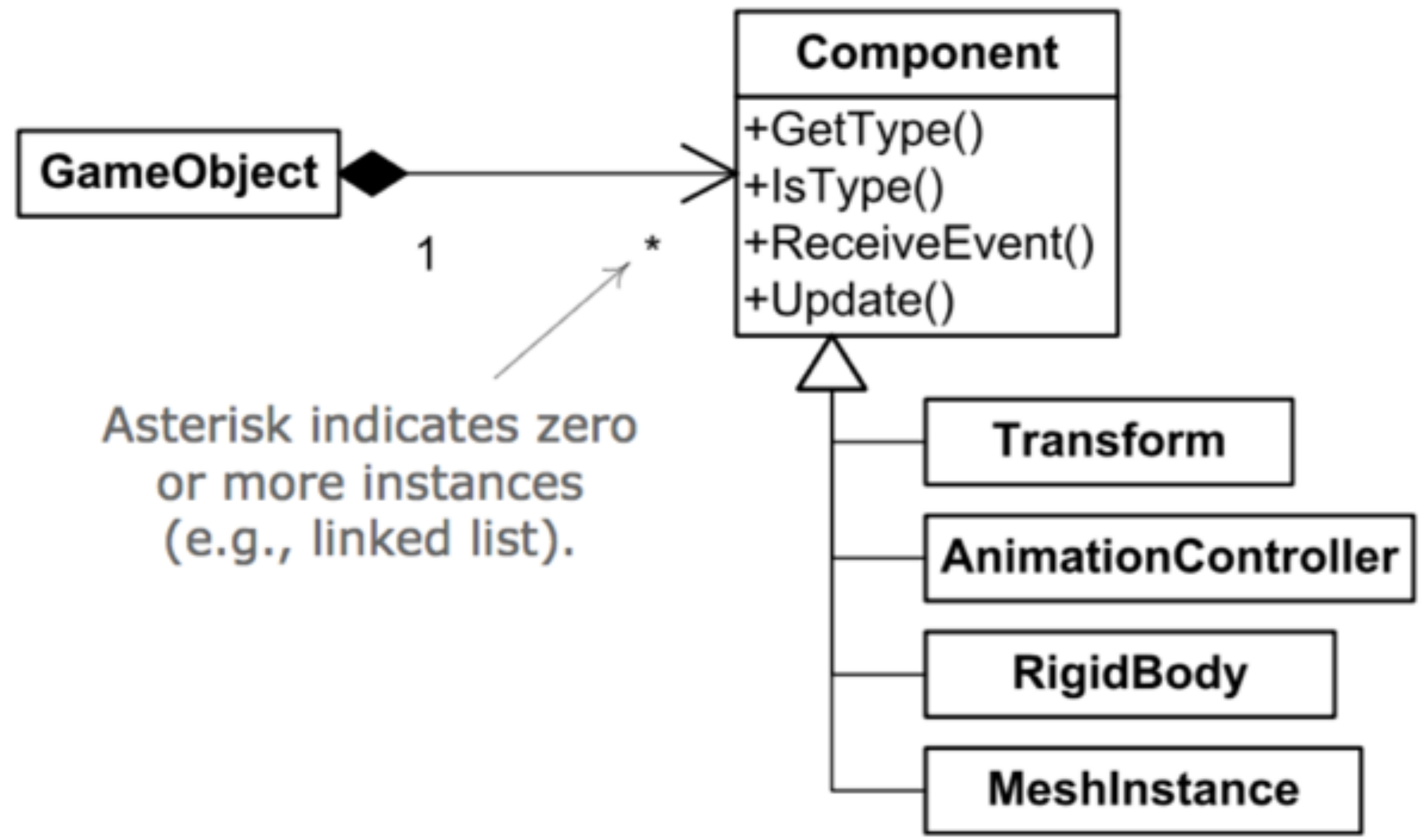


Figure 15.9. A linked list of components can provide flexibility by allowing the hub game object to be unaware of the details of any particular component.

Property-Centric Architecture

- Object1
  - Position (10,2,3)
  - Orientation (0,90,0)
- Object2
  - Position (-15,0,0)
  - Health = 12
- Object3
  - Health = 20

VS

- Position
  - Object1 (10,2,3)
  - Object2 (-15,0,0)
- Orientation
  - Object1 (0,90,0)
- Health
  - Object2 = 12
  - Object3 = 20

Pros and Cons of Property-Centric

- Pros:
  - More memory efficient (no wasted overhead)
  - Continous memory and no fragmentation
- Cons:
  - Hard to figure out the current state of a Game Object (when debugging)
  - It’s hard to enforce relationships between properties and create large scale behaviours

Part 4:

Game World Data Handling

World Chunk Data

- Static data
  - Just a list of Type and Data pairs
- Dynamic data
  - Initial values of the object’s attributes
  - Specification of the object’s type
  - Usually serialized game object descriptions

Instantiating Serialized classes

- In C++ we need to know the class at compile time
- A factory pattern can be applied to get around this issue, using a look up table and a factory class

Spawners and Type Schemas

- Serializing Tool-Side implementation means, we have to know details about the Runtime implementation
- Instead using abstract descriptions we can break that coupling
- Spawners are lightweight data-only representations of a Game Object that contains an id of the object’s Tool-side type.

Object type Schemas

- Type definitions for Game Object types

```
type Light
{
    String       UniqueId;
    LightType    Type;
    Vector       Pos;
    Quaternion   Rot;
    Float        Intensity : min(0.0), max(1.0);
    ColorARGB    DiffuseColor;
    ColorARGB    SpecularColor;
    // ...
}

type Vehicle
{
    String          UniqueId;
    Vector          Pos;
    Quaternion      Rot;
    MeshReference   Mesh;
    Int             NumWheels : min(2), max(4);
    Float           TurnRadius;
    Float           TopSpeed : min(0.0);
    // ...
}
```

```
enum LightType
{
    Ambient, Directional, Point, Spot
}
```

Default Attribute Values

- Instead of having to instantiate all attributes manually on Tool-side, some objects can just use default values
- Especially smart for inherited type definitions, where you just want the default parent values.
- If an attribute is not defined in a spawner, the default value can just be used instead.

Streaming and Loading Game Worlds

- Simple Level Loading (aka. 1 level at a time)
- Air Locks (small chunks, between big chunks)
- Game World Streaming (load multiple chunks, depending on world region)
- Streaming low resolution assets, then bigger ones.

Memory Management for Object Spawning

- Offline Allocation (the figure it out once method)
  - Simply only allow spawning / destroying once
  - No memory fragmentation
  - Designers/Programmers have to guess size ahead of Runtime
- Dynamic Allocation (the do it on the fly method)
  - Use object type pools for repeated objects
  - Alternatively use small object pools of fixed sizes (8, 16, 32, etc.)
  - Or use memory relocation

Saved Games

- Different from level loading, because we already have most data in the world chunks
- We just need to save the state of the objects that are different
- Checkpoint saving uses less memory, because we only need health, ammo, inventory etc. or even less like in Limbo

Exercise:

Creating a Component based GameObject structure