Class Diagrams

Programming – Code Design & Data Structures

Contents

- What is a Class Diagram
- Why Use a Class Diagram
- Modelling:
  - Classes
  - Inheritance
  - Associations
  - Interfaces

What Is a Class Diagram

- Documents the *structure* of a program
- Shows what *types* are being modelled in the system
  - Classes
  - Interfaces
  - Datatypes
- Describes the *relationships* between classes
  - It should be more than just a list of all the classes in your program

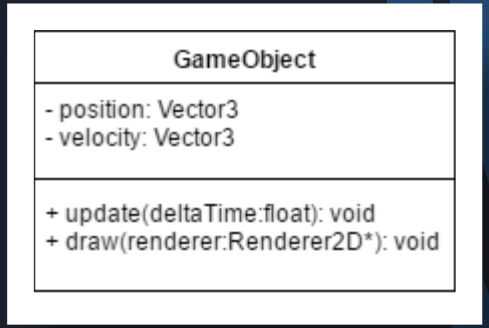
Why Use a Class Diagram

- Developers use class diagrams to document the system's coded or soon to be coded classes

- When you want more than just a conceptual model of your system
  - Although programming language agnostic, class diagrams document design decisions regarding a system's implementation

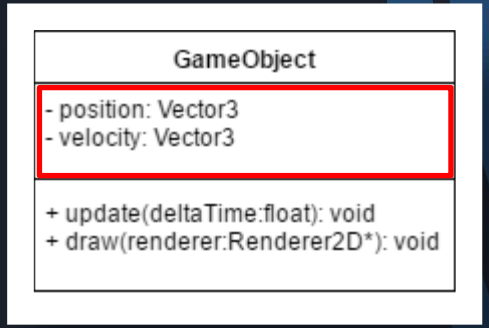
Classes

- Each class is shown in a rectangle containing 3 compartments
  - Top compartment shows class name
  - Middle compartment lists class's attributes
  - Bottom compartment lists class's operations



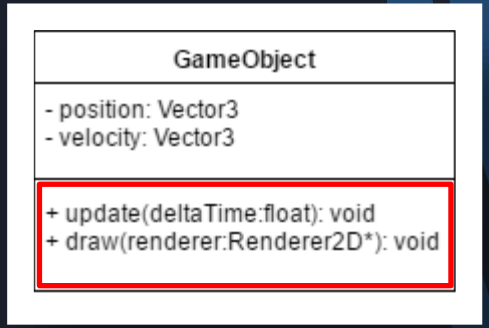
Classes

- Attribute List
  - [+/-] name : attribute type [= default value]
  - The attribute section is optional
  - Each attribute (member variable) listed on a new line
  - Access modifier [+/-]
    - + means public
    - - means private



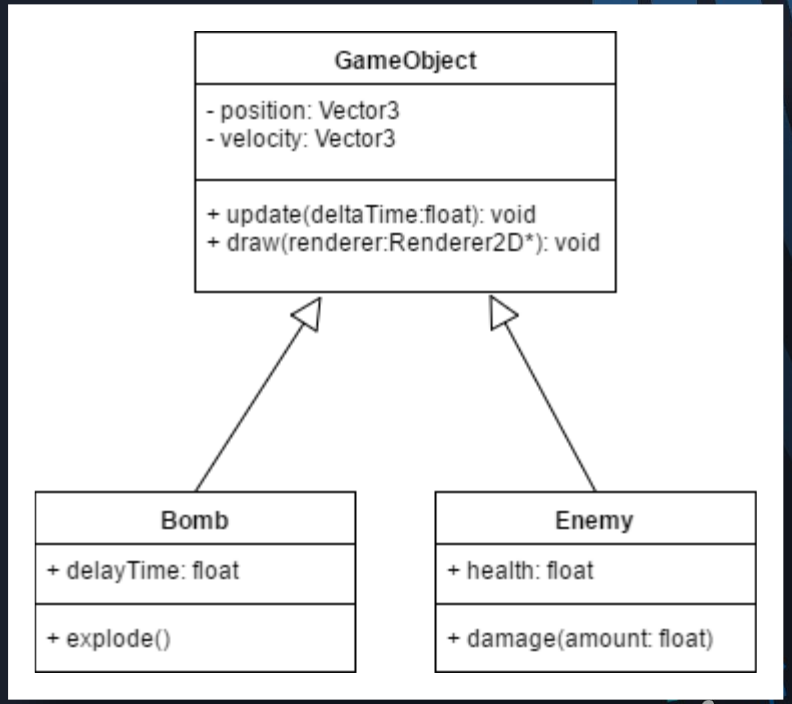
Classes

- Operations List
  - [+/-] name(parameter list): return type
  - Each operation (member function) listed on a new line
  - Access modifier [+/-]
    - + means public
    - - means private
  - Can optionally use ‘in’ and ‘out’ to specify if an argument is for input or output
    - + setPosition(in position:Vector3)
  - Return type can be absent for void functions



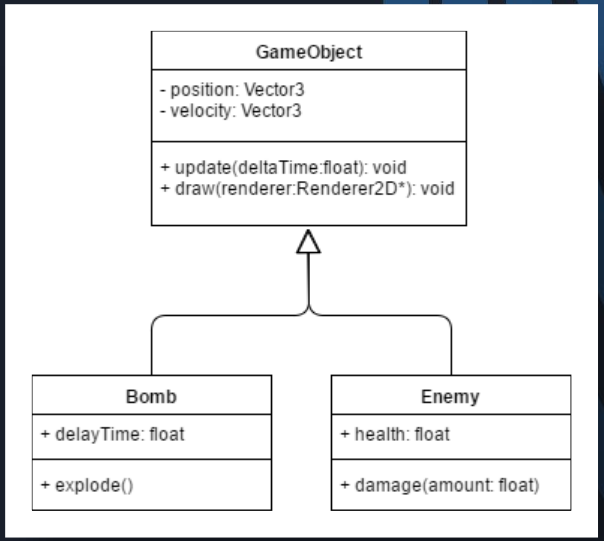
Inheritance

- A *solid line* is drawn from the child class, with a *closed*, *unfilled arrowhead* pointing to the base class



Inheritance

- For inheritance, you can merge lines together like a tree branch
  - This is called *tree notation*



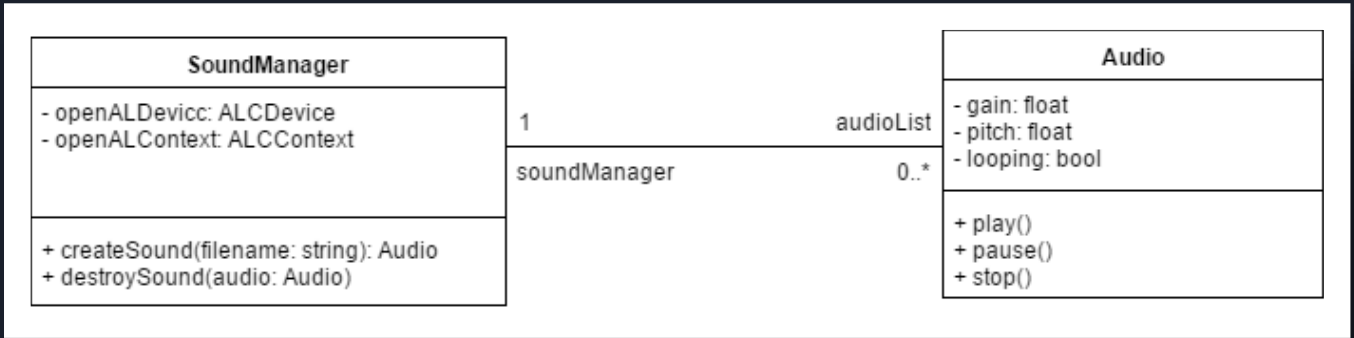
Associations

- Show the relationship between certain objects

- There are 5 kinds of association
  - Bi-directional
  - Uni-directional
  - Aggregation
  - Composition
  - Association Class (not shown)

Bi-Directional (standard) Associations

- A link between two classes
- Associations are always assumed to be bi-directional
  - Means that both classes are aware of the relationship
- *audioList* is a container with one or more *Audio* instances
  - You could also list *audioList* in the attribute list of *SoundManager*, but its inclusion is implied by the association
  - (likewise for the *soundManager* variable in *Audio*)

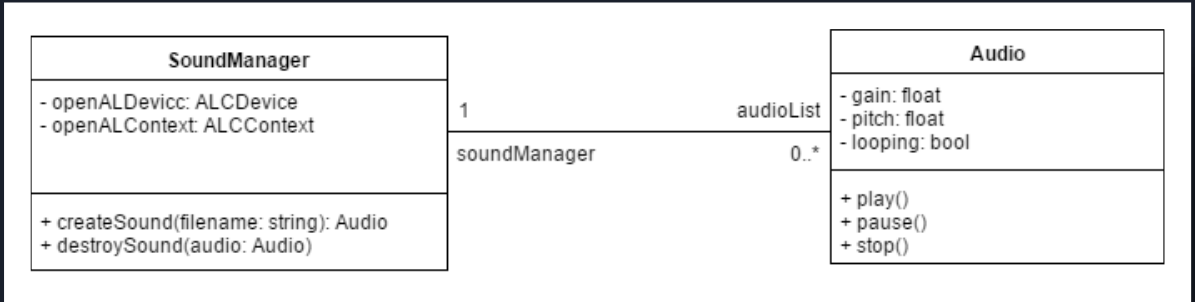


Bi-Directional (standard) Associations

- These are actually not that common in game programming
- They increase coupling, making the system less adaptable to change
- Usually there are better ways to structure the code
  - In the *SoundManager* example, we could call *Play()* on an *Audio* instance, which then calls the appropriate *SoundManager* code to play the sound
  - Refer to the aieBootstrap framework to see the code

Association Multiplicity Values

- In the example, the association has multiplicity values
  - 0..* a *SoundManager* can have 0 or more *Audios*
  - 1 an *Audio* is present in only 1 *SoundManager*

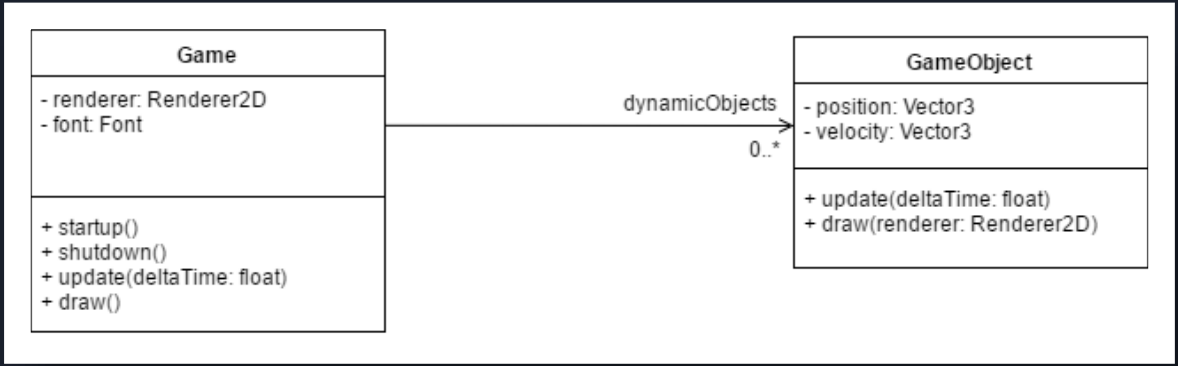


Indicator	Meaning
0..1	Zero or one
1	One only
0..*	Zero or more
*	Zero or more
1..*	One or more
3	Only three
5..15	Five to 15

Uni-Directional Association

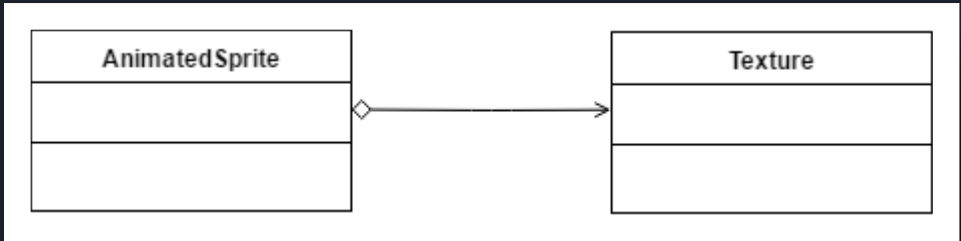
- Two classes are related, but only one knows that the relationship exists

- Drawn as a solid line with an *open arrowhead*, pointing to the known class



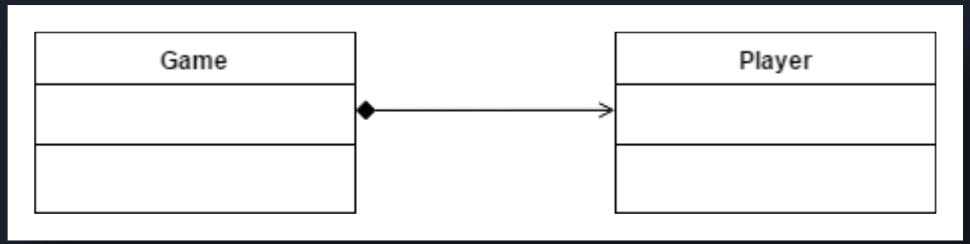
Aggregation

- A special type of association
- Models a lifecycle independence
  - The *part* class can live without the *parent* class
- Draw a solid line from the parent to the part class, with an *unfilled diamond* on the parent's end
- In this example, we could create and destroy the *Texture* independently from the *AnimatedSprite* instance



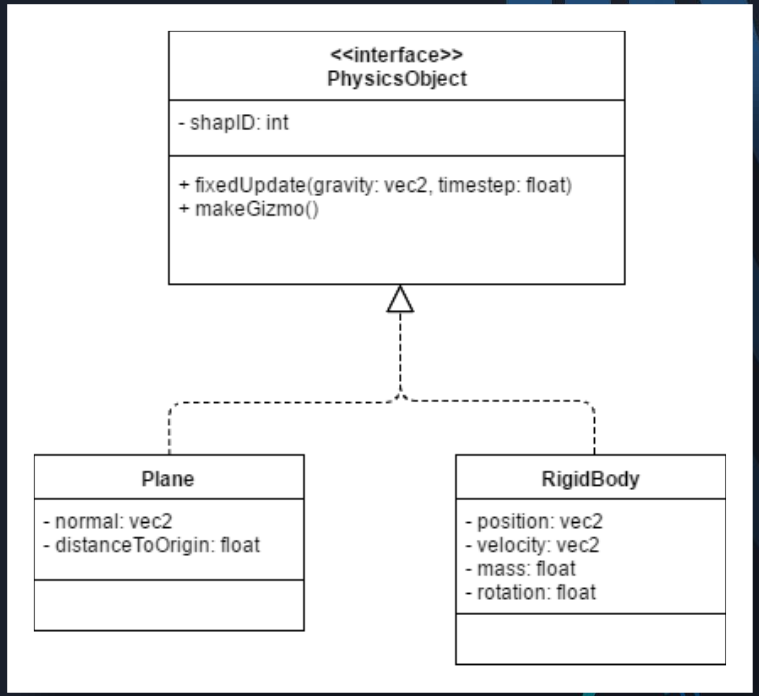
Composition

- A special type of association
- Models a lifecycle dependence
  - The *part* class can *not* live without the *parent* class
  - When the parent is destroyed, the part is also destroyed
- Drawn like aggregation, except the diamond is *filled*



Interfaces

- In C++, these are abstract classes
  - You can not make an instance of the interface
- Instances can only be made of the implementing class(s)
- The dotted line indicates the class *implements* the interface
  - Rather than being a sub-class



How Much Detail?

- It is important that your class diagram is clear and unambiguous
  - Use as much detail as necessary to achieve this
- Remember, this describes how your system should be or is implemented
- Using the wrong symbols can lead to confusion, or to an incorrect or unintended implementation

Summary

- Class Diagrams document the structure of a program
- They show the classes, but more importantly the relationships between classes
- A class's attributes and operations can be described in detail
- Inheritance and association can be modelled to depict specific system architecture

Further Reading

- Donald Bell. 2017. *UML basics: The class diagram*. IBM Developer Works. [ONLINE] Available at: https://www.ibm.com/developerworks/rational/library/content/RationalEdge/sep04/bell/. [Accessed 11 April 2017]

- Scott W. Ambler. 2017. *UML 2 Class Diagrams: An Agile Introduction*. [ONLINE] Available at: http://www.agilemodeling.com/artifacts/classDiagram.htm. [Accessed 11 April 2017]