Design patterns in videogames

Videogames Technology

Universidad de Alcalá ISG

Objectives

- Understand the need of design patterns
- Distinguish the main design patterns categories
- Apply the main patterns to problems in videogames

Bibliography

1. *Desarrollo de Videojuegos, Arquitectura del Motor de Vieojuegos.* UCLM.
2. Wikipedia

Table of Contents

- Software engineering applied to videogames
  - Software engineering
  - Design pattern definition
  - Design pattern structure
- Design patterns
  - Types of design patterns
  - Creational patterns
  - Structural patterns
  - Behavioral patterns
- Videogame models
  - Render loop
  - Game loop
- Game architectures
  - Game architectures
  - Callbacks
  - Events
  - State machine

Software engineering applied to videogames

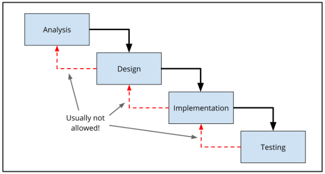
Software engineering (I)

Game programming is a complex task

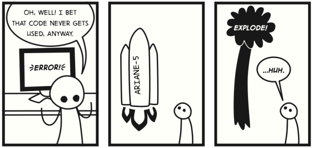
- Rarely done by a single person
- Development team ⇒ Software Engineering

Classic development process (**software lifecycle**)

1. Analysis: What do I need?
2. Design: How do it?
3. Implementation: Do it
4. Testing: Does it work?



The waterfall process



Source: http://www.cosc.canterbury.ac.nz/csfieldguide/SoftwareEngineering.html

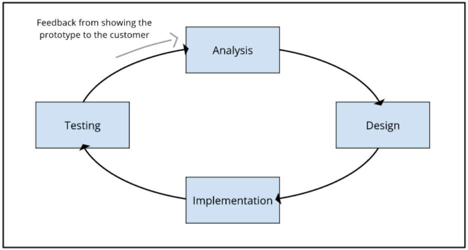
More: http://en.wikipedia.org/wiki/Iterative_and_incremental_development

Software engineering applied to videogames

Software engineering (II)

Many development processes

- Usually, game development is **iterative**



Feedback from showing the prototype to the customer

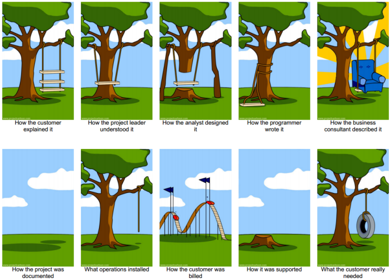
Analysis

Design

Implementation

Testing

Iterative software development



How the customer explained it
How the project leader understood it
How the analyst designed it
How the programmer wrote it
How the business consultant described it
How the project was documented
What operations installed
How the customer was billed
How it was supported
What the customer really needed

Design is critical for the videogame lifecycle: Class hierarchy

Software engineering applied to videogames

Design pattern definition (I)

Some problems happen frequently

- Experience is a valuable asset, but it is not enough
- A design pattern stores knowledge on successful designs

Design pattern

It is the description of the communication among objects and classes customized to solve a generic design problem under a given context

Design Patterns. Elements of Reusable Object-Oriented Software Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (GoF- Gang of Four), 2008

Software engineering applied to videogames

Design pattern definition (II)

Informal definition: A design pattern is a solution to a design problem

- Its utility has been verified by experience
- It must be reusable

More: `http://en.wikipedia.org/wiki/Software_design_pattern`

Software engineering applied to videogames

Design pattern definition (III)

Design patterns goals

- Provide a portfolio of reusable elements in software design
- Avoid loose time searching solutions to already solved problems
- Formalize a shared vocabulary
- Standarize designs
- Ease learning

Design pattern do not want to

- Impose some design alternatives
- Remove designer creativity

Software engineering applied to videogames

Design pattern structure

Four components:

1. **Name**. Short name that identifies the pattern
2. **Problem and context**. Problem that the pattern solves, context where it takes sense and list of *preconditions*
3. **Solution**. General solution not tied to any programming language. Usually described with UML diagrams.
4. **Advantages/drawbacks**.

Additionally:

- Classification, applicability, structure, roles, colaborators, implementation, example code, related patterns, ...

Design patterns

Types of design patterns

Three great groups:

1. **Creational patterns**. Objects and data structures creation
    - Singlenton, factory, abstract factory, ...
2. **Structural patterns**. Class hierarchy, relation and composition of objects
    - Model-View-Controller (MVC), adapter, façade, proxy, ...
3. **Behavioral patterns**. Objects message passing (communication)
    - Observer, chain of responsability, command, iterator, state, strategy, ...

Additional domain patterns

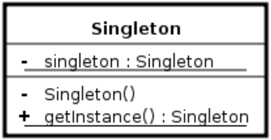
- Web development, GUIs, business, ...

Design patterns

Creational patterns: Singlenton

Singlenton

Problem: Guarantee only one instance of a class
Solution: Private constructor, instanciate the class through a public method
Example: We need only one game instance



Code example

```
public class Singleton {
  private static Singleton INSTANCE = new Singleton();

  private Singleton() {}

  public static Singleton getInstance() { return INSTANCE; }
}
```

Design patterns

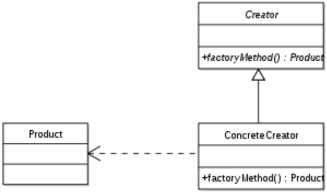
Creational patterns: Factory

Factory

Problem: Create new object

Solution: Group object creation login in a factory class

Example: Create warriors and rogues in a RPG game



Factory code example

```
public class CarFactory {
  public static Car buildCar(String model) {
    switch (model) {
      case "small":
        return new SmallCar();
      case "sedan":
        return new SedanCar();
      case "luxury":
        return new LuxuryCar();
      }
    }
}
```

Design patterns

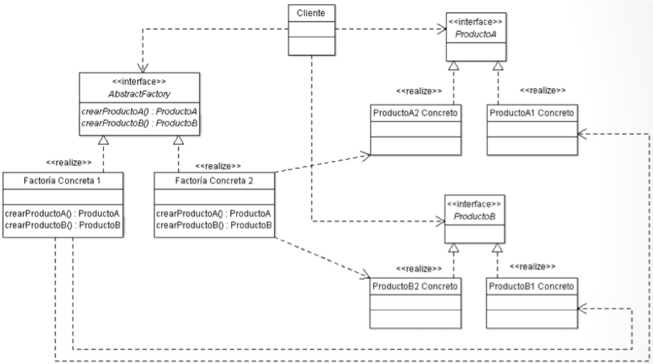
Creational patterns: Abstract factory (I)

Abstract factory

Problem: Create families of new objects

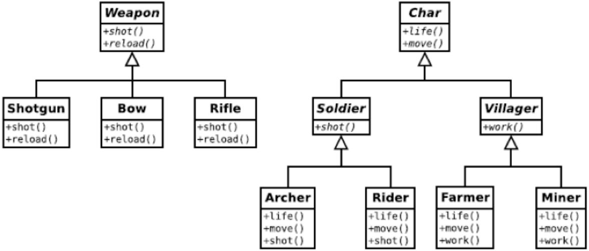
Solution: Create a hierarchy of factories

Example: Create human or orc warriors in a RPG game



Design patterns

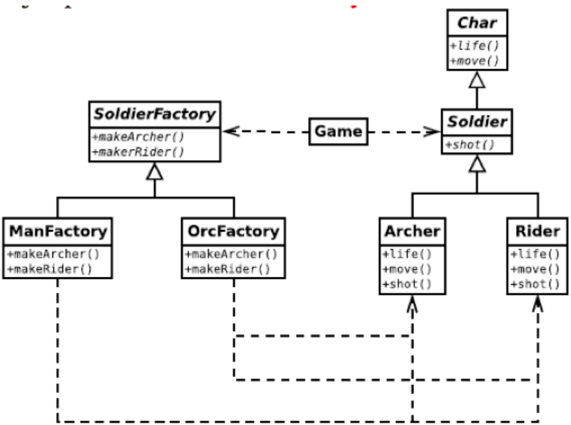
Creational patterns: Abstract factory (II)



RTS game class hierarchy

Design patterns

Creational patterns: Abstract factory (III)



Example of *abstract factory* applied to a RTS game

Design patterns

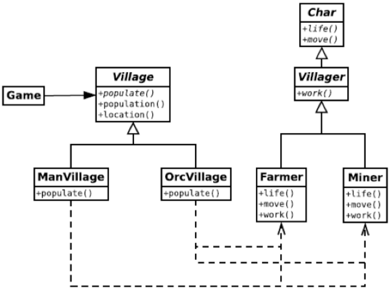
Creational patterns: Factory method

Factory Method

Problem: Create new objects

Solution: Method that instanciates objects

Example: Populate a village with characters



Design patterns

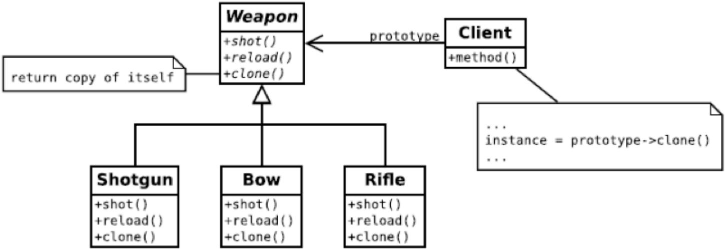
Creational patterns: Prototype

Prototype

Problem: Create a large number of objects whose instantiation is heavy

Solution: Clone objects

Example: Instanciate a large number of weapon objects

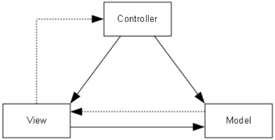


Design patterns

Structural patterns: MVC (I)

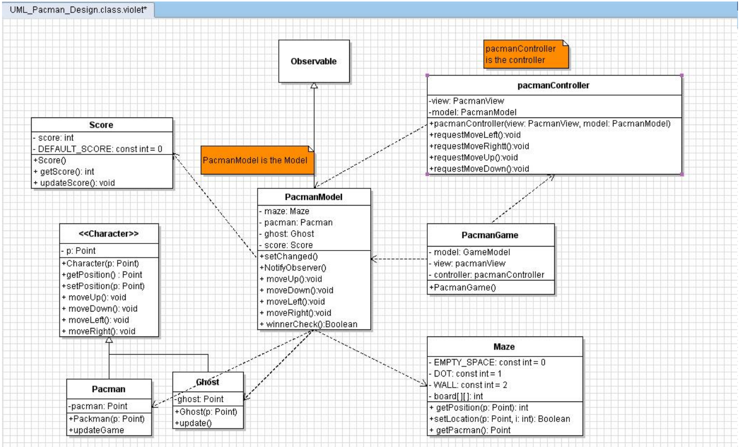
Model-View-Controller (MVC)

Problem: Decouple logic, data and visualization
Solution: Use different classes to contain data, its visualization and the game control
Example: Any game or graphical application



Design patterns

Structural patterns: MVC (II)



UML_Pacman_Design.class.violet*

Observable

pacmanController is the controller

pacmanController
-view: PacmanView
-model: PacmanModel
+pacmanController(view: PacmanView, model: PacmanModel)
+requestMoveLeft():void
+requestMoveRightt():void
+requestMoveUp():void
+requestMoveDown():void

Score
- score: int
- DEFAULT_SCORE: const int = 0
+Score()
+ getScore(): int
+ updateScore(): void

PacmanModel is the Model

PacmanModel
- maze: Maze
- pacman: Pacman
- ghost: Ghost
- score: Score
+setChanged()
+NotifyObserver()
+ moveUp():void
+ moveDown():void
+ moveLeft():void
+ moveRight():void
+ winnerCheck():Boolean

<<Character>>
- p: Point
+Character(p: Point)
+getPosition() : Point
+setPosition(p: Point)
+ moveUp(): void
+ moveDown(): void
+ moveLeft(): void
+ moveRight(): void

PacmanGame
- model: GameModel
- view: pacmanView
- controller: pacmanController
+PacmanGame()

Maze
- EMPTY_SPACE: const int = 0
- DOT: const int = 1
- WALL: const int = 2
- board[][]: int
+ getPosition(p: Point): int
+setLocation(p: Point, i: int): Boolean
+ getPacman(): Point

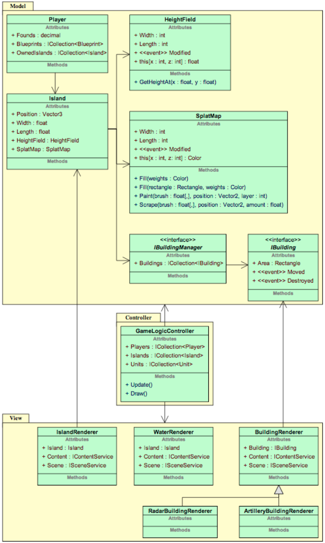
Pacman
-pacman: Point
+Packman(p: Point)
+updateGame

Ghost
-ghost: Point
+Ghost(p: Point)
+update()

Source: https://code.google.com/p/pacpounder/downloads/list

Design patterns

Structural patterns: MVC (III)



Source: `http://blog.nuclex-games.com/2010/09/mvc-in-games/`

Design patterns

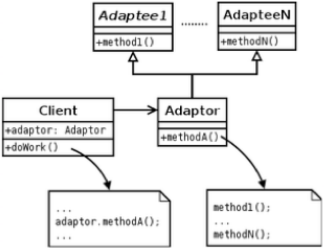
Structural patterns: Adapter

Adapter

Problem: One class needs to invoke a method in another class, but it cannot

Solution: Use an intermediate class with a new interface

Example: Incompatible third-party library



Design patterns

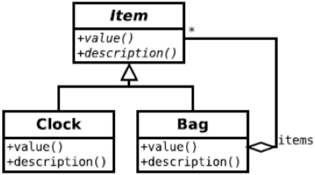
Structural patterns: Composite

Composite

Problem: Store objects that might contain other objects

Solution: Objects composition

Example: Game whose player keeps an inventory whose items might contain other items



Design patterns

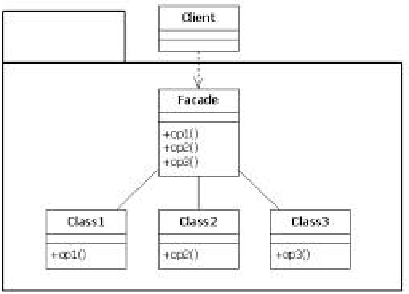
Structural patterns: Façade

Façade

Problem: Complex interface to a set of classes

Solution: Create an intermediate class that simplifies the interface

Example: Graphical library with several operation modes



Design patterns

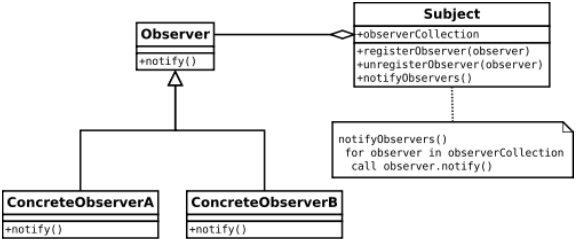
Behavioral patterns: Observer (I)

Observer

Problem: Notify a set of objects when another object changes

Solution: Link a set of observers to an observed object

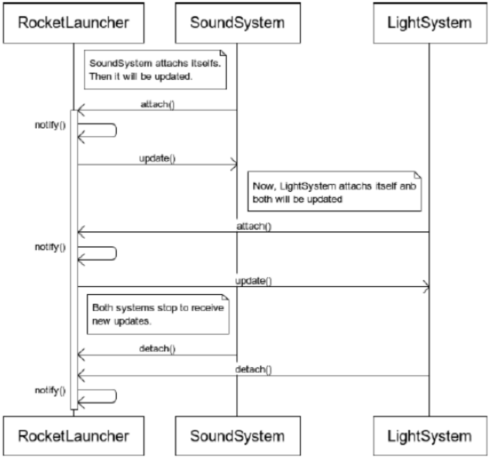
Example: A view that has to know when the model changes



```
notifyObservers()
 for observer in observerCollection
  call observer.notify()
```

Design patterns

Behavioral patterns: Observer (II)


RocketLauncher
SoundSystem
LightSystem
SoundSystem attachs itselfs. Then it will be updated.
attach()
notify()
update()
Now, LightSystem attachs itself anb both will be updated
attach()
notify()
update()
Both systems stop to receive new updates.
detach()
detach()
notify()
RocketLauncher
SoundSystem
LightSystem

Design patterns

Behavioral patterns: Observer (III)

DataStore.java

```java
public class DataStore extends Observable {
  private String data;

  public String getData() { return data; }

  public void setData(String data) {
    this.data = data;
    setChanged();
    notifyObservers();
  }
}
```

Screen.java

```java
public class Screen implements Observer {
  @Override
  public void update(Observable ob, Object arg) {
    // Do something
  }

  public static void main(String args[]) {
    Screen screen = new Screen();
    DataStore datastore = new DataStore();
```

Design patterns

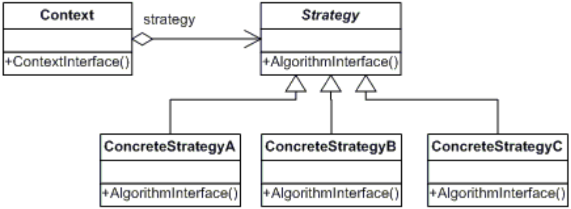
Behavioral patterns: Strategy (I)

Observer

Problem: Choose in execution time which method use from several ones

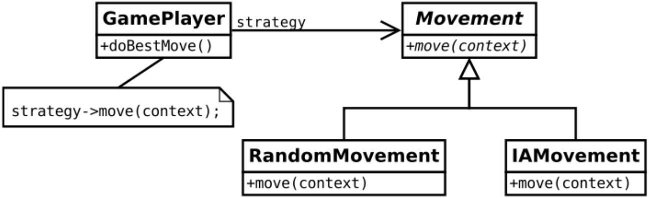
Solution: Encapsulate the method in a class

Example: A fighter with several fighting styles



Design patterns

Behavioral patterns: Stategy (II)



Design patterns

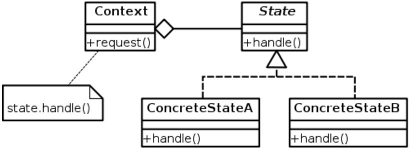
Behavioral patterns: State (I)

State

Problem: Implement a state machine

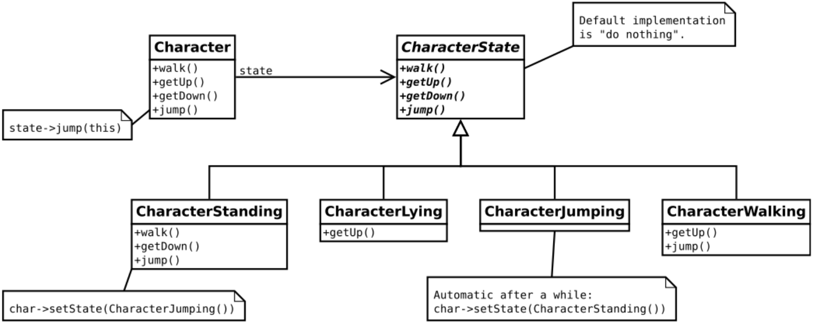
Solution: Encapsulate state transitions

Example: NPC behavior



Design patterns

Behavioral patterns: State (II)

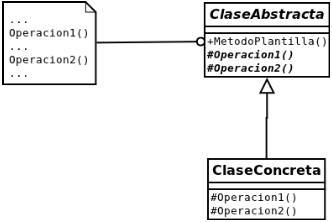


Design patterns

Behavioral patterns: Template method (I)

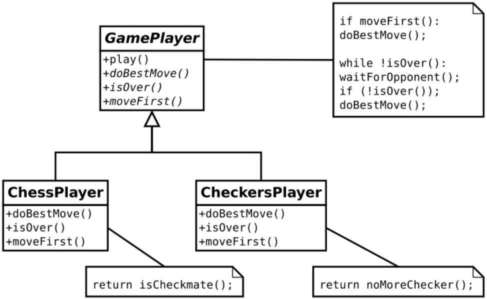
Template Method

Problem: Customize an algorithm
Solution: Divide the algorithm in methods that can be overriden
Example: Chess and checkers games



Design patterns

Behavioral patterns: Template method (II)



```
if moveFirst():
doBestMove();

while !isOver():
waitForOpponent();
if (!isOver());
doBestMove();
```

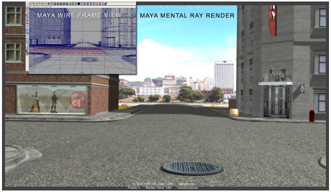
```
return isCheckmate();
```

```
return noMoreChecker();
```

Videogame models

Render loop (I)

- The render loop handles visualization and rendering
- Objectives in 2D games
  - Minimize pixels to draw: Draw only those pixels that have changed
  - Maximize fps



Render example

- Objectives in 3D games
  - Camera uses to change everytime: The same technique cannot be used
  - Minimize the number of primites to draw in each iteration of the render loop

Videogame models

Render loop (II)

Render loop

```
while (true) {
  // Update camera, usually according to a
  // predefined path
  updateCamera();

  // Update position, orientation and rest
  // of the state of the entities in the game
  updateSceneEntitites();

  // Render a frame in a buffer
  renderScene();

  // Interchange the buffer to visualize the image
  swapBuffers();
}
```

Info: http://wiki.wxwidgets.org/Making_a_render_loop

Videogame models

Game loop (I)

The main element in a videogame is the game loop

- It is the main control structure in the game
- It controls its execution
- It handlers the transitions among states
- The game loop independizes the game execution from the hardware

Classical programs only reacts with user actions

- Videogames are always performing an action
- Game loop implements this easely
- The game engine contains the game loop

Videogame models

Game loop (II)

- There are many subsystems in a videogame
  - Rendering engine (render is to generate an image)
  - Collition detection
  - Collition handling
  - AI subsystem
  - Game subsystem
- Most of these subsystems require periodic updates
- The most critical one is the animation system
  - Frequency: 30 or 60 Hz
  - Syncronized with the rendering subsystem
  - Objective: Provide a good fps rate to generate a realistic experience
- Not all the components are so strict, for instance, AI

Videogame models

Game loop (III)

- There are several ways to implement the game loop
- The easiest one is to have several loops within the game loop
  - Render loop
  - AI loop
  - Multimedia loop
  - Iteration loop

Basic game loop

```
boolean running = true;

while (running) {
  updateGame();
  displayGame();
}
```

Game loop

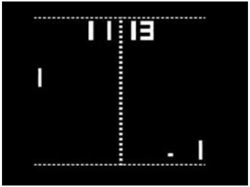
```
while(running) {
  checkUserInput();
  runAI();
  moveEnemies();
  resolveCollisions();
  drawGraphics(); //Render loop
  playSound();
}
```

Videogame models

Game loop (IV)

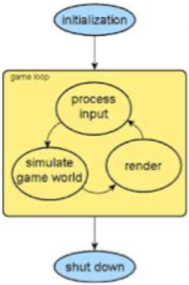
```
int main (int argc, char* argv[]) {
  init_game(); // Game initialization

  while (1) {  // Game loop
    capture_events(); // Capture events
    if (exitKeyPressed()) break; // Exit
    move_paddles(); // Update paddles
    move_ball();  // update ball
    collision_detection();
    if (ballReachedBorder(LEFT_PLAYER)) {
      score(RIGHT_PLAYER);
      reset_ball();
    }
    if (ballReachedBorder(RIGHT_PLAYER)) {
      score(LEFT_PLAYER);
      reset_ball();
    }
    render();
  }
}
```

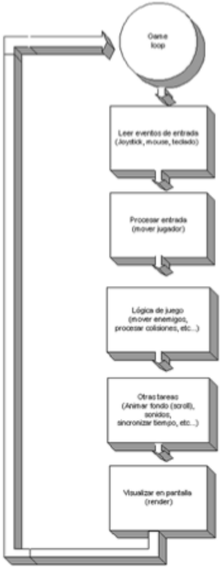
Pong game loop example

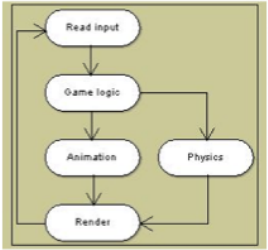


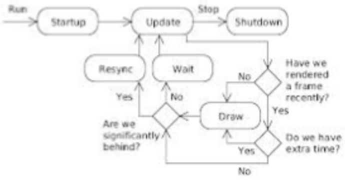
Videogame models

Game loop (V)




Game loop
Leer eventos de entrada (Joystick, mouse, teclado)
Procesar entrada (mover jugador)
Lógica de juego (mover enemigos, procesar colisiones, etc...)
Otras tareas (Animar fondo (scroll), sonidos, sincronizar tiempo, etc...)
Visualizar en pantalla (render)





Videogame models

Game loop (VI)

The game loop depends on the platform

- DOS games and some consoles are designed to exploit computational resources
- PC games depend on limitations imposed by the OS
- Games use to be multithreaded to exploit multicore machines

Game architectures

Game architectures

Game loop can be implemented in different ways

- Architectures based on callbacks
- Architectures based on events
- Architectures based on state machines

Most of them implement one or more control loops

Game architectures

Callbacks (I)

- **Callbacks**: Code that is executed to handle an event
  - Function or object
  - Callbacks are used to ``fill" source code
- Related term: **framework**
  - Application partially completed that the developer has to complete

Videogame models

Callbacks (II)

```
void update (unsigned char key, int x, int y) {
  Rearthyear += 0.2;
  Rearthday += 0.2;
  glutPostRedisplay();
}
// More code
int main (int argc, char** argv) {
  glutInit(&argc, argv);

  glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE);
  glutInitWindowSize(640, 480);
  glutCreateWindow("Session #04 - Solar System");
  // Define callbacks
  glutDisplayFunc(display);
  glutReshapeFunc(resize);
  glutKeyboardFunc(update);

  glutMainLoop();
  return 0;
}
```

Game architectures

Events

- An event represents a change in the game state
- Two types
  - **External**: Generated by the interactions
    Example, The player press a key or moves the joystick
  - **Internal**: Generated by the game logic
    Example, NPC respawn
- Most game engines include an event subsystem
  - Closely related to the *Observer* pattern

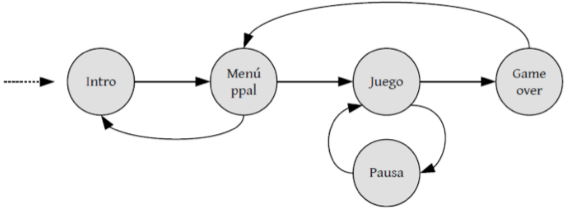
Game architectures

State machine

A game goes through a number of states

- Introduction
- Main menu
- Game
- Game over

State machine: A set of states and transitions



Warning: State machines play a mayor role in game AI