# Java game programming

# Game engines

2010

Fayolle Pierre-Alain

# Plan

- Some definitions
- List of (Java) game engines
- Examples of game engines and their use

# A first and simple definition

- A game engine is a (complex) software system designed for the creation and development of video games
- It abstracts the (platform dependent) details of doing common game-related tasks
- Its core functionality (may) include: a rendering engine, a physics engine, a scene graph, a sound engine …

# Purpose of a game engine

- To provide a flexible and reusable software platform for game development
- It does so by providing all or part of the core functionality needed for a game
- This helps reducing:
  - The complexity of game development
  - The time-to-market
    - Time-to-market is the length of time it takes from the conception of a product to its availability for sale
  - The cost of development

# Hardware / Platform abstraction

- Game engines usually provide platform abstraction (for example: game consoles and personal computers, Windows or Linux)
- Rendering engines are built upon graphics API such Direct3D or OpenGL providing a software abstraction to different video cards
- Input / sound engines may be built upon low level libraries such as DirectX or SDL providing software abstraction to input devices, network cards, or sound cards

# Hardware abstraction

- Prior to hardware accelerated graphics, software renderers were used
- Software renderers may still be present in some rendering engines, e.g. to emulates features absent from some cards
- Physics engines may build on physics processing units (PPU) and their API and provide an abstraction of the PPU hardware

# Component-based architecture

- Game engines are often designed with a component-based architecture that allows part of the system to be replaced with other components
- A component is a software package or a module encapsulating a set of functions (and / or data)
- Components communicate with each other through interfaces

# History of game engines

- Historically games were designed from the ground up to make optimal use of limited hardware
- Early games for consoles and personal computers were mostly programmed in low-level language (assembly or C) and optimized for a particular hardware
- The term “game engine” gained popularity in the 90s with games such as Doom or Quake

# History of game engine

- The concept of game engine was made popular with first person shooting (FPS) game where engine (program) and assets (texture, characters, etc) became clearly separated
- It allowed the same engine to be re-used with different scenarios and assets to produce different games
- Popular example is the Quake engine which was used in Quake, Half-life, Hexen II and other

# Trends in game engine

- FPS games saw lots of progress on rendering engines: from wireframes to 3D world, (basic) graphics hardware acceleration, popularization of GPU, lighting and pixel shaders
- Progress on physics engine as well: rigid body dynamics, soft body dynamics (cloth), fluid dynamics
- New hardware / platform targets: mobile phones
- Usage of higher-level languages: C#, Java …

# Middleware

- A middleware is a software that resides between applications and the underlying operating systems, network protocol stacks, and hardware
- Middleware's role is to functionally bridge the gap between lower-level and applications
- Originally game engines were developed internally for reuse in future game of the company
- Then some companies started to sell them to provide another source of income (Id Software)
- Now some companies specialize only in developing and selling game engines (or part of it); the term middleware is used in this context

# Example: DMM Engine

- Middleware physics engine developed by Pixelux
- Simulates a large sets of physical properties by using Finite Element Analysis instead of the classical rigid body kinematics
- Based on an algorithm developed by Prof. James O’Brien in his PhD thesis: “Graphical modeling and animation of fracture” (Berkeley)

# Taxonomy of game engines

- Game engines can be classified using different criteria:
  - The type of game that can be developed with (FPS, MMORPG, simulation, …)
  - The functionalities that are covered (rendering, physics)
  - The platform that are supported (Linux / Windows, game console / personal computer)
  - Commercial / Open source

# List of (Java) game engine

- List limited to: open-source game engines and either written in Java language or providing bindings to the Java language
- Written in Java:
  - Ardor3D (http://www.ardor3d.com/)
  - jMonkeyEngine (http://www.jmonkeyengine.com/home/)
  - Jogre (http://jogre.sourceforge.net/main.htm)
  - Lightweight Java Game Library (http://www.lwjgl.org/index.php)

# Game engines with Java bindings

- Typically written in C++
- Third party bindings for the Java language are provided (typically using JNI: Java Native Interface)
- Irrlicht Engine (http://www.irrlicht3d.org/wiki/)
- Crystal space (http://www.crystalspace3d.org/main/Main_Page)

# JOGRE

- Java Online Gaming Real-time Engine
- http://jogre.sourceforge.net/main.htm
- A client / server game engine allowing the creation of multi-player online games (ex: chess, go, …)
- Provide a set of packages to help developers with: rendering, network and communications, state manipulations
- Provide base abstract classes to be extended by developers

# LWJGL

- LightWeight Java Game Library
- http://www.lwjgl.org/index.php
- Provide developers access to cross-platform libraries for:
  - Rendering (using OpenGL)
  - Sound (using OpenAL and FMOD)
  - Input controls (gamepad, joystick)
  - Timer

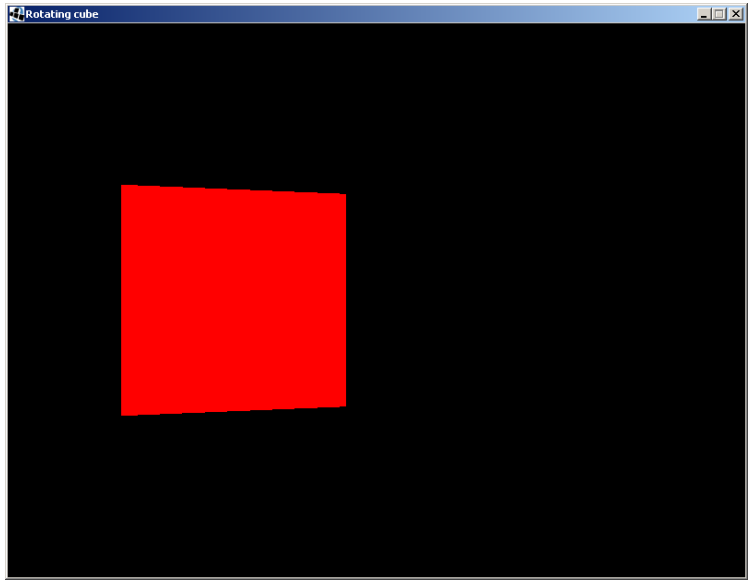
# LWJGL

- Low-level
- Similar to JOGL (for OpenGL) and JOAL (for OpenAL)
- Regroup these low-level libraries together into one library
- Main packages: input, OpenGL, and OpenAL
- Style is procedural (in spirit of OpenGL and OpenAL which are C libraries)
- Best to use as a component for other game engines
  - For example: jMonkeyEngine or ardor3d use LWJGL in their rendering engine

# LWJGL

- opengl.Display and opengl.DisplayMode: to browse available modes, select one, toggle between full-screen and windowed mode …
- Opengl.GL11: implementation of OpenGL 1.1 specifications; render in 2D and 3D, texture mapping, transformations
- Input.Keyboard: methods to check if a key is pressed, key codes (similar classes for mouse and joysticks)
- Util.Timer to control the game speed

# Spinning cube




```
public static void main(String args[]) {
    cube cube1 = new cube();
    cube1.run();
}

public void run() {
    // within a try {} catch() {`}
    init();
    while (!done) {
        keyLoop();
        render();
        Display.update();
    }
}
```

# Spinning cube

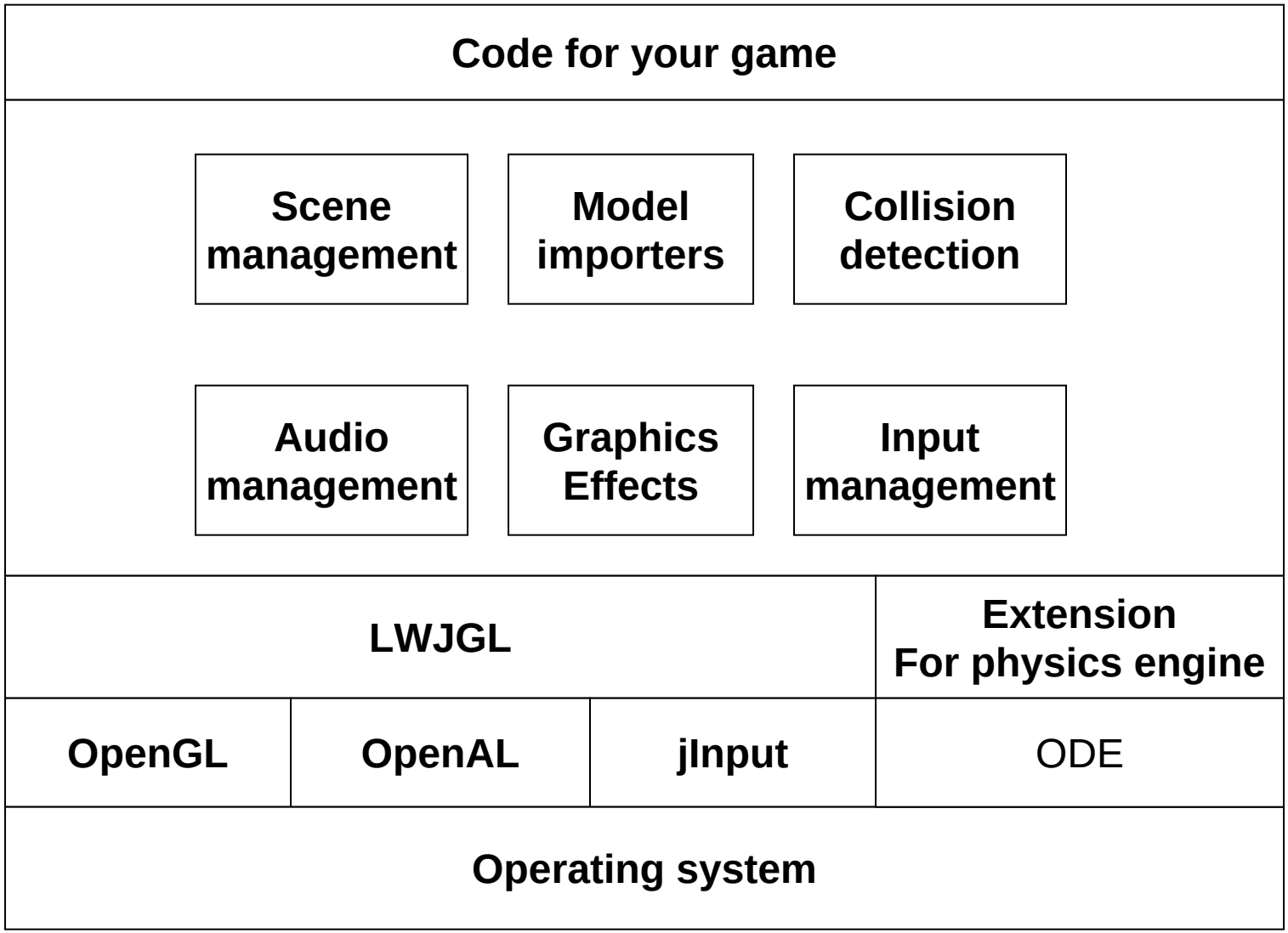
```
private void init() throws Exception {
    // 1. Init display
    Display.setFullscreen(fullscreen);
    …..
    // Loop to find an acceptable display
        displayMode = …..;
    …..
    Display.setDisplayMode(displayMode);
    Display.create();

    // 2. Init GL
    GL11.glShadeModel(…);
    …..
}
```

```
private void keyLoop() {
  …..
  if(Keyboard.isKeyDown(Keyboard.KEY_RIGHT)){
    angleY = 0.015f;
  }
  …..
}
```

# jMonkeyEngine

- http://www.jmonkeyengine.com/home/
- jME is a Java based 3D game engine
- It provides a scene graph architecture
- It provides functionalities for sound, input, GUI dev., and several effects (water, particle system, …)
- It is mostly written in Java except for a thin JNI layer used to interface with audio, video and input device

# Architecture

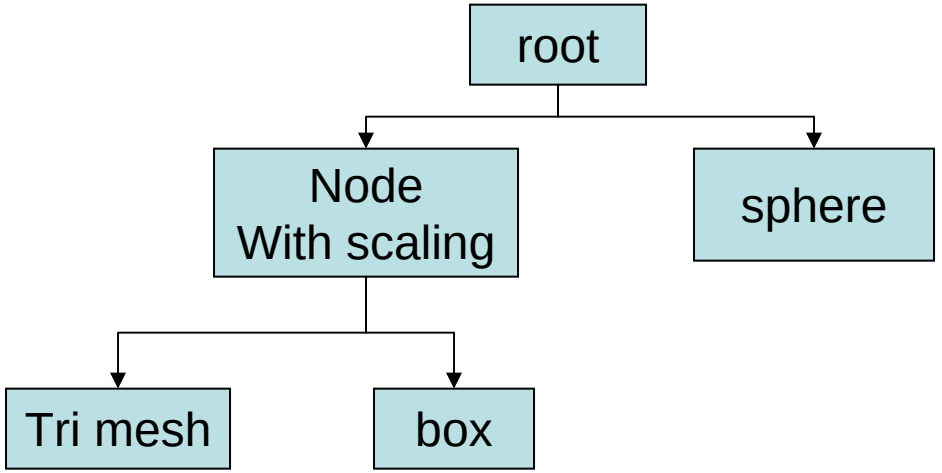


# jME renderer

- It uses OpenGL (LWJGL) or software rendering
- Responsibilities:
  - Transform from world space to screen space
  - Traverse the scene graph and renders visible portions of it

# Scene graph

- A hierarchical data structure (tree) used to group data
- Internal nodes used to apply operations (geometrical transformation, lighting, …)
- Leaf nodes contain geometrical data



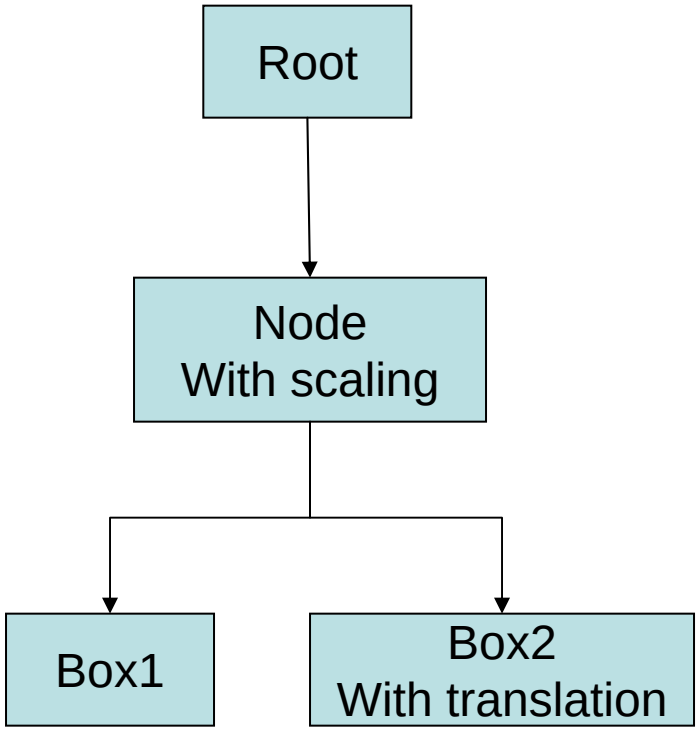
# Benefits of a scene graph

- Simplify management of attributes:
  - Ex: define a light applying only to a sub-tree
- Facilitate the definition of hierarchical geometrical models
  - In OpenGL, it would be done by using glPushMatrix() and glPopMatrix() on the ModelView matrix
- Grouping objects in a region
  - Help in culling non-visible areas of the scene

# jME scene graph

- Leaf nodes: geometrical data
- Internal nodes maintain spatial and semantical information
  - Transformation: for positioning, scaling and orienting objects
  - Bounding volumes: for culling and intersection testing
  - Render state: information passed to the renderer for rendering object (ex: lighting state, texture state, material state)
  - Animation state: to represent time varying node data

# Example




```
protected void simpleInitGame() {
    Node n = new Node("Node1");

    Box b1 = new Box("Box1", new Vector3f(0, 0, 0),
                             new Vector3f(1, 1, 1));
    b1.setModelBound(new BoundingBox());
    b1.updateModelBound();

    Box b2 = new Box("Box1", new Vector3f(0, 0, 0),
                             new Vector3f(1, 1, 1));
    b2.setModelBound(new BoundingBox());
    b2.updateModelBound();
    b2.setLocalTranslation(1.0f, 2.0f, 0.0f);

    n.attachChild(b1);
    n.attachChild(b2);
    n.setLocalScale(2.0f);

    rootNode.attachChild(n);
}
```

# jME sound

- Sounds are defined in a similar way as the geometry inside the scene graph
- Sounds are rendered in 3D (using OpenAL – JOAL)
- Access to the audio system is performed by: *AudioSystem audio = AudioSystem.getSystem();*
- Possibility to load .ogg or .wav files to be played during the game

# Graphics functionalities

- Loading, mapping and rendering of textures
- Various 3D model data loaders:
  - MD2: Quake 2 model format
  - ASE: 3D Studio Max Ascii format …
- Built-in primitives: point, line, sphere, box, triangle meshes, …
- States: lights, textures, fog, materials, shading, alpha

# Effects

- Particles, water, terrain, shadow
- Example: Particle system
  - Creation through the factory:
    ParticleMesh p = ParticleFactory.buildParticles(“particles”, 60);
  - Setup properties: size, color, …
    p.setInitialVelocity(0.1f);
    p.setStartSize(3f);
  - Add influences such as wind, gravity, …
    ParticleInfluence wind = SimpleParticleInfluenceFactory.*createBasicWind*(.6f, **new** Vector3f(0, 1, 0), **true**, **true**);
    p.addInfluence(wind);

# Simple example of game

- Framework:
  - jME proposes some application classes:
    - AbstractGame, SimpleGame, SimplePassGame, StandardGame
  - We will use the simplest one: SimpleGame

```
public class TestGame extends SimpleGame {
  @override
  protected void simpleInitGame() { … }
}
```

# jME application classes

- AbstractGame
  - Basic API for games
  - Some implementations common to all game:
    - initSystem();
    - initGame();
    - the game loop which calls update(float) and render(float);
    - cleanup()
- SimpleGame extends BaseSimpleGame
- BaseSimpleGame provides implementation for most of the needed tasks; it is only required to build a scene graph in simpleInitGame()
- AbstractGame -> BaseGame -> BaseSimpleGame

# jME application classes

- SimplePassGame: same as BaseSimpleGame with multi-pass rendering management
- StandardGame:
  - implements basic functionalities needed in a game

# Simple example of game

- jME’s geometrical primitive for the game elements:

```
ball = new Sphere(“Ball”, 10, 10, 1);
ball.setModelBound(new BoundingSphere());
ball.updateModelBound();

player = new Box(“Player”, new Vector3f(), 5, 5, 10);
player.setModelBound(new BoundingBox());
player.updateModelBound();
player.setLocalTranslation(110f, 0f, 0f);

// walls and bricks are similarly made of Box

rootNode.attachChild(ball);
rootNode.attachChild(player);
```

# Input control

- Use KeyBindingManager to register actions and map keyboard input these actions

```
simpleInitGame() {
  KeyBindingManager.getKeyBindingManager().set(“PLAYER_MOVE_LEFT”,
       KeyInput.KEY_1);
}

simpleUpdate() {
  if (KeyBindingManager.getKeyBindingManager().isValidCommand(
       “PLAYER_MOVE_LEFT”, true)) {
     // move the position of the player to the left
     player.getLocalTranslation().z -= speed * timer.getTimePerFrame();
  }
}
```

# Ball movement

- Use the ball speed and the time elapsed (between the last and the current call) to update the ball position at each update

```
simpleUpdate() {
    ball.getLocalTranslation().x = ball.getLocalTranslation().x
                                   + ball_speed.x * timer.getTimePerFrame();
    ball.getLocalTranslation().y = ball.getLocalTranslation().y
                                   + ball_speed.y * timer.getTimePerFrame();
    ball.getLocalTranslation().z = ball.getLocalTranslation().z
                                   + ball_speed.z * timer.getTimePerFrame();
}
```

# Collision detection

- We use the bounding volume only for checking collision

```
simpleUpdate() {
  if (player.hasCollision(ball, false)) {
    ball_speed.x = ball_speed.x * (-1.0f);
  }

  // same for the back wall

  if (lateral_walls.hasCollision(ball, false)) {
    ball_speed.z = ball_speed.z * (-1.0f);
  }
}
```

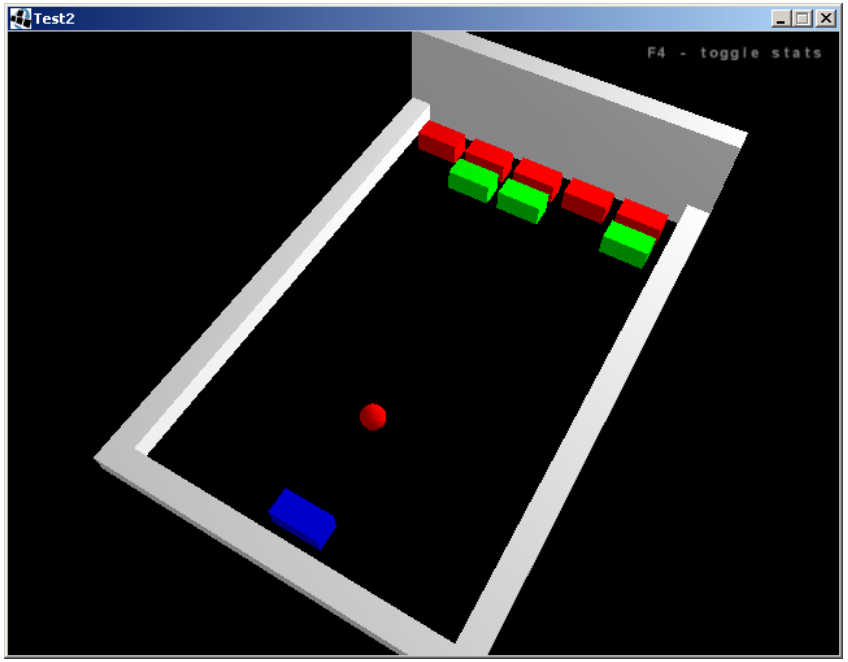
# Collision with the bricks

- For the bricks in addition to changing the ball velocity direction, we need to remove the brick from the scene graph

```
simpleUpdate() {
  if (!brick_wall.isEmpty()) {
    for (Brick b : brick_wall) {
      if (!b._removed && b._box != null
          && ball.hasCollision(b._box, false)) {
            ball_speed.x = ball_speed.x * -1.0f;
            rootNode.detachChildNamed(b._name);
            b._removed = true;
      }
    }
  }
}
```

# Simple example of game

Breakout like game



# Adding sound to the game

- First obtain access to the audio system:
  AudioSystem audio = AudioSystem.getSystem();
- Setup a track in the init part:
  AudioTrack collide_sound = audio.createAudioTrack(“collision.ogg”, false);
- Every time a collision is detected, play the sound:
  collide_sound.play();
- Update the AudioSystem in the game loop:
  AudioSystem.getSystem().update();

# More effects

- For example: add some particle system making some firework effect once the game is finished
- Add texture to the player paddle, to the walls, …
- Add a background music
- All of these can be quickly done using the jME API

# Physics engine

- Collision detection is already provided
- Cloth simulation is also provided (jmex.effects.cloth.ClothUtils and jmex.effects.cloth.CollidingClothPatch)
- For doing more physically based simulation: possibility to add jME physics 2 that provides access to ODE, PhysX

# Summary: game engines

- Game engines are software libraries bringing (through API) to the developer functionalities needed for developing games
- As with any other software libraries, they allow the developer to re-use components and decrease its development time
- Java game engines developed in Java (with some JNI layers) or bindings to game engines (developed in other languages)
- Examples with LWJGL (low-level) and jMonkeyEngine (higher level) showed how game engines can facilitate the creation of game

# Summary: Java for game development

- Cross platform
- Power of Java technology:
  - Easy to use
  - Size of the standard library
  - Object oriented
- Deployment
  - Applet or application
  - Full-screen or windowed
- Bottlenecks can be rewritten in lower level languages (with default implementation in Java) for speed improvement (JNI, Swig)