11 Design Patterns for Multimedia Programs

11.1 Specific Design Patterns for Multimedia Software

11.2 Classical Design Patterns Applied to Multimedia

Literature:

R. Nystrom: Game Programming Patterns, genever banning 2014,
See also http://gameprogrammingpatterns.com/

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides,
Design Patterns, Addison-Wesley 1994
Design Patterns

• A design pattern is a generic solution for a class of recurring programming problems
  – Helpful idea for programming
  – No need to adopt literally when applied

• Origin:
  – Famous book by Gamma/Helm/Johnson/Vlissides ("Gang of Four", "GoF")
    » List of standard design patterns for object-oriented programming
    » Mainly oriented towards graphical user interface frameworks
    » Examples: Observer, Composite, Abstract Factory

• Frequently used in all areas of software design

• Basic guidelines:
  – Patterns are not invented but recovered from existing code
  – Pattern description follows standard outline
    » E.g.: Name, problem, solution, examples
Window Place: Architectural Pattern

Christopher Alexander et al., A Pattern Language, 1977 (quoted in Buschmann et al. 1996)

• **Problem:** In a room with a window and a sitting opportunity users have to decide whether to have a look or to sit.

• **Solution:**
  At least one window of the room shall provide a sitting place.

• **Structure:**

Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.

*Christopher Alexander et al., A Pattern Language*
Description of a Design Pattern

- Name
- Problem
  - Motivation
  - Application area
- Solution
  - Structure (class diagram)
  - Participants (usually class, association and operation names):
    » Role name, i.e. place holders for parts of implementation
    » Fixed parts of implementation
  - Collaboration (sequence of events, possibly diagrams)
- Discussion
  - Pros and cons
  - Dependencies, restrictions
  - Special cases
- Known uses
Patterns for Multimedia Software

- The following examples of patterns are not taken from literature, but derived from the material in this lecture
  - Based on various platforms, also older ones
- Types of patterns:
  - Cross-platform patterns
  - Patterns specific for a certain platform
Classification Space

Time usage:
- Still picture
- Linear progress
- Interaction dependent progress

Space usage:
- Static layout
- Scenes
- Scenes & objects
- Fully dynamic

Degree of interactivity
- Fully automatic (passive)
- Confirmations & questions (reactive)
- Interactive controls (proactive)
- Creation & dragging of objects (directive)

Cross-Platform Multimedia Pattern: Clockwork

- The current properties of presentation elements are derived from the current value of a “clock” ticking at regular time intervals
- Time usage: Linear progress
- Limited interactivity: Automatic or confirmations & questions
- Usually combined with static layout or scenes and objects
- Examples:
  - Timeline in Flash, Director, JavaFX
  - EnterFrame-Events in Flash ActionScript
  - Ticking scripts in Squeak
  - Clock class in PyGame
  - Scheduler in Cocos2d-x

```java
PActivity flash =
    new PActivity(-1, 500, currentTime + 5000) {
        protected void activityStep(long elapsedTime) {
            ...
        }
    }
```

University of Maryland “Piccolo” framework (see cs.umd.edu/hcil/piccolo)
Cross-Platform Multimedia Pattern: Interpolation

• A parameter (usually regarding a graphical property) is assumed to change its value continuously dependent of another parameter (e.g. time). The dependency can follow a linear or other rules of computation.
  – Fixed values for the dependent parameter are given for certain values of the base parameter.
  – Intermediate values of the dependent parameter are computed by interpolation.

• Space usage: scenes&objects mainly

• Time usage: Linear progress only

• Usually combined with low interactivity (on this level)

• Examples:
  – Tweening in Flash
  – Actions in Cocos2d-x
  – JavaFX transitions

```java
PActivity a1 = aNode.animateToPositionScaleRotation(0, 0, 0.5, 0, 5000);
```
Cross-Platform Multimedia Pattern: Scene Graph

- Graph structure for all represented objects
- Space usage: Scenes & objects or fully dynamic
- Time usage: Linear progress or interaction dependent
- Examples:
  - Scene graph of Cocos2d-x, JavaFX
  - Scene graph of Piccolo
  - Implicit: Film Explorer view in Flash

"SceneBeans"
Cross-Platform Pattern: Time Container Algebra

- Presentation is built from atomic parts (processes) each of which is executed in a *time container*.
- Time containers are composed by algebraic operations: sequential composition, parallel composition, repetition, mutual exclusion, synchronization options
- Time usage: Linear progress
- Space usage: Scenes or scenes & objects
- Low interactivity
- Examples:
  - SMIL body: seq, par, excl
  - Cocos2-x actions: Sequence, Spawn, Repeat
  - Animations class of “JGoodies” animation framework for Java
  - Sequence of frames and parallelism of layers in Flash
Various Syntactical Representations for a Single Concept

```cpp
Sprite* imageSprite r1;
auto act1 = ... frame1;
auto act2 = ... frame2;
r1->runAction(
    Sequence::create(act1, act2, NULL));
```

**Cocos2d-x**

**Authoring Tool (Flash-like)**
11 Design Patterns for Multimedia Programs

11.1 Specific Design Patterns for Multimedia Software
11.2 Classical Design Patterns Applied to Multimedia

Literature:

R. Nystrom: Game Programming Patterns, genever banning 2014,
See also http://gameprogrammingpatterns.com/

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides,
Design Patterns, Addison-Wesley 1994
GoF Structural Pattern: Composite

• Situation:
  – Complex tree-structured objects

• Motivation:
  – Use homogeneous interface for objects and compositions thereof

Exercise (Cocos2d-x):
Compare with classes
– Node
– Scene
– Layer
– Sprite
– DrawNode
GoF Patterns: Observer, Template Method, Façade

- Observer (behavioral):
  - Decoupling event handling from event sources
  - Notification interface for observers, registration functions for subjects
  - *Basic principle of all event handling mechanisms*
    *Built into most GUI and multimedia frameworks*

- Template Method (behavioral):
  - Modifying detailed steps in an algorithm keeping its general structure
  - Program skeleton with calls to abstract methods (detail steps)
  - Subclasses specialize the abstract methods
  - *Basis for inversion of control in frameworks*
    (e.g. createScene() and init() methods in Cocos2d-x)

- Façade (structural):
  - Maintaining a simple interface object to a large body of code
  - Used e.g. in Cocos2d-x in the Director object for game setup
GoF Behavioral Pattern: Command (1)

- Command:
  - Encapsulate a request as an object, alternative to callbacks
- Example:

```cpp
void InputHandler::handleInput()
{
    if (isPressed(BUTTON_X)) jump();
    else if (isPressed(BUTTON_Y)) fireGun();
    else if (isPressed(BUTTON_A)) swapWeapon();
    else if (isPressed(BUTTON_B)) lurchIneffectively();
}
```
GoF Behavioral Pattern: Command (2)

InputHandler

- buttonX
- buttonY
- buttonA
- buttonB

handleInput()

Command

{abstract}

execute() {abstract}

JumpCmd

execute()

void execute() {
    jump()
}

FireCmd

execute()

void execute() {
    fire()
}

What are potential advantages of this restructuring?
Generalizing Command

• Commands may be parameterized
  – Example: Actors

```cpp
class JumpCommand : public Command {
    public:
        void execute(GameActor& actor) {
            actor.jump();
        }
}
```

• Application example:
  – Defining Non-Player Characters (NPCs)
  – Typically based on "AI" (Artificial Intelligence)
  – AI code emits Command objects
Using Command for Undo Functionality

• Undo/Redo:
  – Generally important for all software – basic ingredient for good usability
  – Games: Important mainly in game creation, e.g. Level Design Software
• Extend Command by inverse to `execute()` method:

```
Command
{abstract}

execute() {abstract}
undo() {abstract}
```

• Various application variants:
  – Including game replay
GoF Structural Pattern: Flyweight (1)

- Think about a majestic old growth forest
  - Consisting of individual trees…
  - Enormous amount of resource needs!

"Use sharing to support large numbers of fine-grained objects efficiently."

(GoF Book)
GoF Structural Pattern: Flyweight (2)

• Basic idea: Separate between
  – Data shared by many instances
  – Data specific for a single instance

• Goal: Memory and time efficiency, only
GoF Creational Pattern: Prototype (1)

- Assuming we need many instances of a few types of objects:

- Extending the root class with a `clone()` method:

```
Monster
  – speed
  – health
  – ...

... clone()
```
GoF Creational Pattern: Prototype (2)

• Generic "spawner" (creator) object
  – Holds a local "prototype"
  – Creates on request copies of the prototype

• Spawner clones not only structure but also state!
  – May be used to differentiate more fine-grained than class hierarchy
GoF Creational Pattern: Singleton

- Realization of "single-instance" classes:
  - Only one instance exists and is created if needed
  - Access is homogeneous, whether instance exists or not
- Attention: To be called in a mutually exclusive way in multi-threaded applications!
- Examples for application in games:
  - Director class in Cocos2d-x (Cocos2::Director::getInstance())
  - Audio engine in various frameworks (CocosDenshion::SimpleAudioEngine::getInstance())
Criticism on Singleton Pattern

• Global state information:
  – Makes it difficult to reason about code locally (theoretical and practical aspects)
  – Introduces hidden coupling between code parts
  – Is not concurrency-friendly

• Multiple instances may be useful (e.g. for logging)

• Simple static methods of a static class may be more flexible

• General advice: Do not apply design patterns everywhere possible!
• Specific advice: Limit application to a few system-global assets
GoF Structural Pattern: State

- Name: **State**
- Problem:
  - Flexible and extensible technique to change the behavior of an object when its state changes.
- Solution :

```
Context
request()
setState (s: State)

State
{abstract}
handle() {abstract}

State1
handle()

State2
handle()

... calls state.handle
```
Example for State (1)

```
void Heroine::handleInput(Input input) {
    switch (state_) {
    case STATE_STANDING:
        if (input == PRESS_B) {
            state_ = STATE_JUMPING;
            yVelocity_ = JUMP_VELOCITY;
            setGraphics(IMAGE_JUMP);
        }
        break;
    case STATE_JUMPING:
        if (input == PRESS_DOWN) {...
        break;
    case STATE_DUCKING:
        if (input == RELEASE_DOWN) {...
        break;
    }
}
```

Classical implementation
Example for State (2)

```
... change state to State_Jumping ...
yVelocity_ = JUMP VELOCITY;
setGraphics(IMAGE_JUMP);
```
Changing States Through Context

• Context of state-changing object:
  – Keeps variable `currentState` of type `State`
  – Provides a static single instance for each of the available states
    e.g. `stateStandingInstance`, `stateJumpingInstance`
  – May provide a special interface for updating the `currentState`

• In all State subclasses:
  – State change by somehow changing the context, e.g.
    `currentState = stateJumpingInstance`
  – Either access to global variable or usage of special interface

• Advantages:
  – Programming language & compiler check for coverage of all possible inputs in any state
  – Clear path for extension, protected by language/compiler checks
Test for Extensibility

- Adding a “pressA” input leading to a “paused” state
- First step: Change the State interface

\[ handlePressA() \]

\[ \rightarrow \] Compiler checks completeness of transitions

- All additional code is concentrated in one class = one file
  - for new state paused, e.g. class State_Paused

- Please note:
  - Transitions (e.g. commands through buttons, external events) are mapped to State interface
  - States (e.g. standing, ducking, jumping, diving, paused) are mapped to subclasses of State