GAME PROGRAMMING PATTERNS

COMMAND PATTERN

A **behavioral** pattern in which an object is used to encapsulate all information needed to perform an action or trigger an event at a later time.

class Command {

public:

virtual void execute() = 0;

virtual void undo(); // optional

} **;**

Example: *binding actions to buttons*

*Y = fireGun()*

*B = lurch()*

*A = swapWeapon()*

*X = Jump()*

class JumpCommand : public Command {

public:

virtual void execute() {

// jump logic

}

};

class InputHandler {

public:

void handleInput() {

if (isPressed(\_buttonX)) {

\_buttonX->execute();

} else if (isPressed(\_buttonY)) {

\_buttonY->execute();

}

// etc…

}

private:

Command\* \_buttonX;

Command\* \_buttonY;

Command\* \_buttonB;

Command\* \_buttonA;

};

The command pattern can also implement *undo* & *redo* functionality. Imagine it keeps a track of state using a stack:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Older | cmd | cmd (undo) | cmd (current) | cmd (redo) | newer |

Class Command {

Public:

Virtual void undo() = 0;

Virtual void redo() = 0;

Virtual void execute() = 0;

};

Each child will keep track of the previous state. Every new command will need to be pushed to the stack. A linked list may also work well here.

FLYWEIGHT

A **structural** design pattern which minimses memory usage by sharing as much data as possible with other similar objects.

For example; imagine objects of type Tree, where all have a mesh, position, leaves texture, bark texture, etc. Instead of creating instances for every tree, you can create a TreeModel class with all the similarities that trees share, and all instances can use this one pointer object for shared data.

class TreeModel {

private:

Mesh mesh;

Texture bark;

Texture leaves;

};

Class Tress {

Private:

TreeModel\* model; // a pointer to the same 1 instance for every tree

};

OBSERVER

A **behavioral** pattern which defines a one-to-many dependency relationship.

class Observer{

public:

virtual void update(Subject\* theChangedSubject) = 0;

};

class Subject {

private:

Observer\* observers;

public:

virtual void attach(Observer\*);

virtual void detach(Observer\*);

virtual void notify(); // loop through observers and invoke update()

};

PROTOTYPE

A **creational** pattern used when the type of objects to create is determined by a prototypal instance, which is cloned to produce new objects.

For example; imagine a game with monsters, and spawners spawn monsters of each type.

e.g. Monster < Ghost, Demon, Sorcerer

Spawner < GhostSpawner, DemonSpawner, SorcererSpawner

class GhostSpawner : public Spawner {

public:

virtual Monster\* spawn() {

return new Ghost();

}

};

This is inefficient! Every monster needs its own separate spawner class.

Instead…

class Ghost : public Monster {

private:

int \_health;

int \_speed;

public:

Ghost(health, speed) : \_health(health), \_speed(speed) {}

virtual Monster\* clone() { return new Ghost(\_health, \_speed); }

};

// just one spawner for all monsters

class Spawner {

private:

Monster\* \_prototype;

Public:

Spawner(Monster\* prototype) : \_prototype(prototype) {}

Monster\* spawnMonster() { return \_prototype->clone(); }

};

Monster\* ghostPrototype = new Ghost(15, 3);

Spawner\* ghostSpawner = new Spawner(ghostPrototype);

FINITE STATE MACHINE

A finite-state-machine (fsm), is used to describe the processes during which information or tasks move from one state to another action.

For example; imagine an input handler:

handleInput(e) {

if (e == keyB) {

if (!jumping) {

jump();

}

} else if (e == keyD) {

If (!jumping && !ducking) {

duck();

} else {

// blah blah blah

}

}

// this can get messy!!

};

This pattern can generally be fixed using fsm.

Use fsm when:

1. You have a fixed set of states
2. You can only be in one state at a time

For example; imagine your player, who can duck, jump, stand and dive.

* Stands by default
* Ducked when down arrow pressed
* Exits duck state when down arrow released
* Jumps when B button pressed from standing state
* Dives when down button pressed from jumping state

class Hero {

private:

HeroState\* \_state;

Public:

Virtual void handleInput(Input input) {

State->handleInput(\*this, input);

}

Virtual void update() {

State->update(\*this);

}

};

Class HeroState {

Public:

Virtual void handleInput(Hero& hero, Input input) {}

Virtual void update(Hero& hero) {}

};

class JumpingState : public HeroState {

private:

int \_chargeTime;

public:

virtual void handleInput(Hero& hero, Input input) {

if (input == keyDown) {

// change to diving state

}

}

Virtual void update(Hero& hero) {

If (\_chargeTime > MAX\_CHARGE) {

hero.pose();

}

}

}

It is common for our states to have onEnter() and onExit() methods.

SUBCLASS SANDBOX

Define behavior in a subclass using a set of operations provided by its base class.

A base class defines an abstract *sandbox method* and several provided *operations*. Each derived sandbox subclass implements the *sandbox method* using the provided *operations*.

For example; imagine super powers, they are all different but also similar. E.g. they may touch the audio system, particle system, transform, etc.

class SuperPower{

protected:

virtual void activate() = 0; // sandbox method

void move(double x, double y, double z) { // code here }

void playSound(SoundId sound) { // code here }

void spawnParticles(ParticleType type, int count) { // code here }

};

class SkyLaunch: public SuperPower {

protected:

virtual void activate() {

move(0, 0, 20); // spring into air

playSound(SOUND\_JUMP);

spawnParticles(PARTICLE\_DUST, 10);

}

};

This is all that is needed in subclasses. The base class should have all the logic.