Lecture 7

Game Programming Patterns

Assoc. Prof. Dr. Burkay Genç

25 Nis, 2024

Game Programming Patterns

Main Resource

These slides are based on the excellent book on Game Programming Patterns by Robert Nystrom.

Architecture, Performance and Games

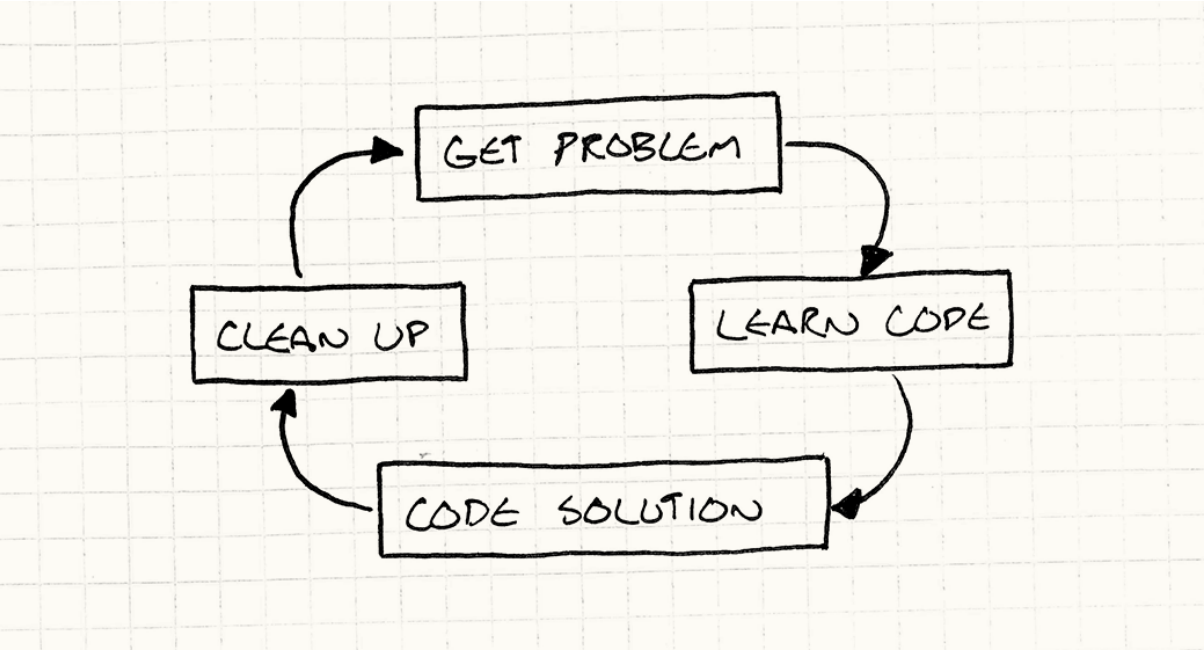
- Patterns are about *organizing* code, not about *writing* code.

Software Architecture

- *Good design* is about *changes to the code*
- If your code will never be changed, then its design is irrelevant
  - If it works, it works

> The measure of a design is how easily it accommodates changes.

How do you make a change?



- Define roblem
- Understand what the existing code is doing
- Come up with a solution
- Clean up your mess
- Repeat

Decoupling

If two pieces of code are coupled, it means you can't understand one without understanding the other.

- So, **decoupling** is to make sure that code pieces are as *independent* from each other as possible.
  - Allows you to *learn less* before you come up with a solution to your problem.
  - Also allows you to change a piece of code without *necessitating* a change to other pieces.
- Decoupling requires good architecture
- Good architecture requires discipline and effort
- Patterns provide the required discipline

YAGNI

You Aren't Gonna Need It

- Constructing a good architecture is very nice
- But, don't overdo it!
- Do not fill your code with unnecessary abstractions, inheritance structures, virtual methods, patterns etc.
  - It takes you forever to trace through all of that scaffolding to find some real code that does something

Performance and Speed

- Performance is about *optimization*
  - Optimization is about *knowing your limits*
- Good architecture is about *flexibility*
  - Flexibility is about *removing your limits*
- So, is it possible to write code that has good architecture and performance at the same time?

There's no easy answer here. Making your program more flexible so you can prototype faster will have some performance cost. Likewise, optimizing your code will make it less flexible.

It's easier to make a fun game fast than it is to make a fast game fun.

Prototyping

- **Prototyping** — slapping together code that's just barely functional enough to answer a design question
- Big caveat

> **Boss**: "Hey, we've got this idea that we want to try out. Just a prototype, so don't feel you need to do it right. How quickly can you slap something together?"
> **Dev**: "Well, if I cut lots of corners, don't test it, don't document it, and it has tons of bugs, I can give you some temp code in a few days."
> **Boss**: "Great!"

A few days later…

> **Boss**: "Hey, that prototype is great. Can you just spend a few hours cleaning it up a bit now and we'll call it the real thing?"

Striking a balance

We have a few forces in play:

1. We want *nice architecture* so the code is easier to understand over the lifetime of the project.
2. We want *fast runtime performance*.
3. We want to *get today's features done quickly*.

Simplicity

- Simplicity is the perfect way to balance architecture and performance
- Simple code is easier to learn and modify
- Simple code is usually faster to run as there is less overhead
- However, simple code is not easy to come up with
  - It requires a lot of knowledge, practice, and *effort*

> "I would have written a shorter letter, but I did not have the time."
>
> Blaise Pascal

> "Perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away."
>
> Antoine de Saint-Exupery

Command Pattern

Command Pattern

- One of the more useful patterns.
- GoF defines it as

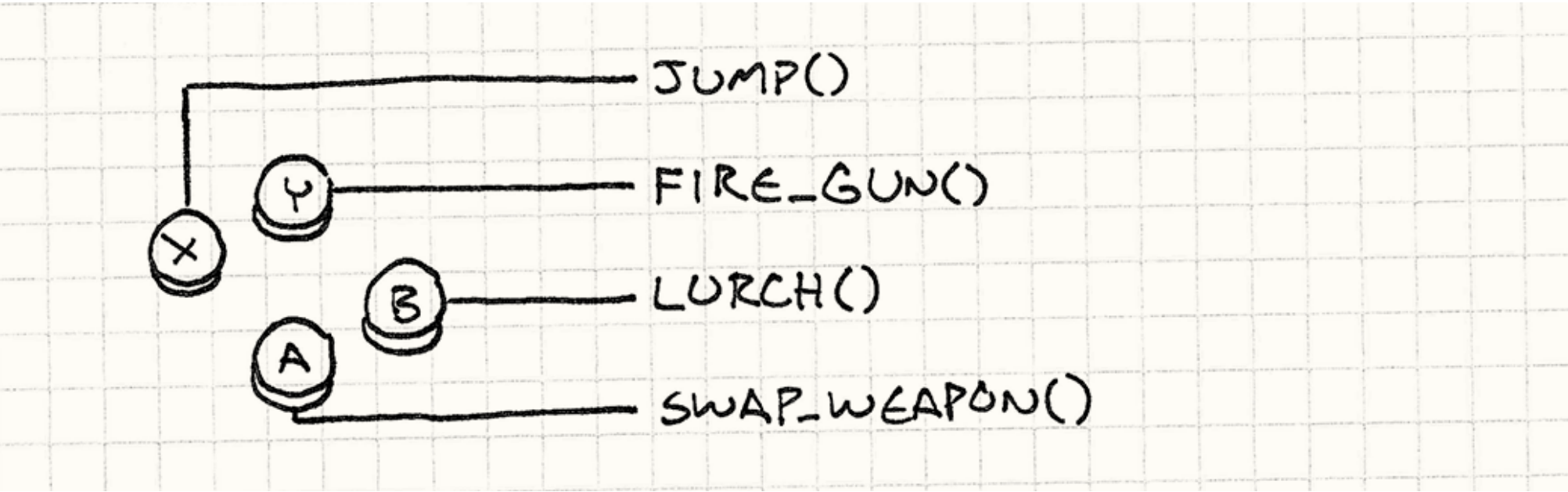
> Encapsulate a request as an object, thereby letting users parameterize clients with different requests, queue or log requests, and support undoable operations.

- Nystrom's definition:

> A command is a reified method call.

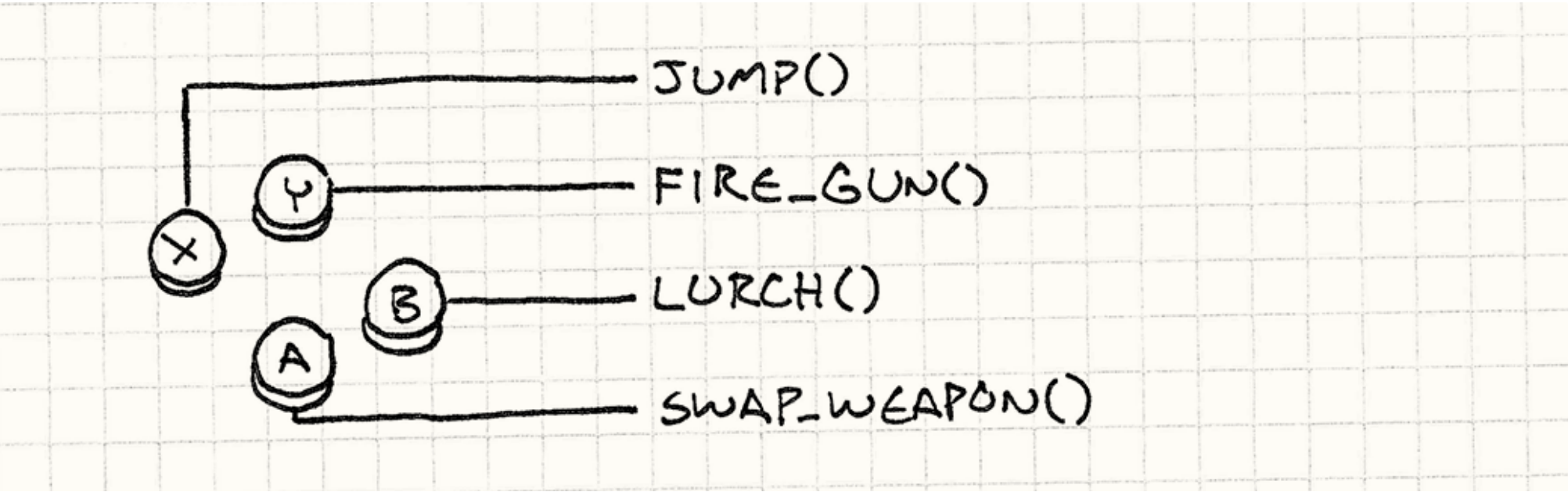
- reify -> make something real, *thingify*, objectify
- wrapping a function call in an object

Example : Configuring input



- Somewhere in every game is a chunk of code that reads in raw user input
  - button presses,
  - keyboard events,
  - mouse clicks
- It takes each input and translates it to a meaningful action in the game

Example : Configuring input




```
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();
}
```

- hardwired user inputs
- no chance to reconfigure input

Command Class

```
class Command
{
  public:
    virtual ~Command() {}
    virtual void execute() = 0;
};
```

- We need to turn those direct calls to `jump()` and `fireGun()` into something that we can swap out
- We need an object that we can use to represent a game action.
- We define a base class that represents a triggerable game command:

Command Pattern

Then we create subclasses for each of the different game actions:

```
class JumpCommand : public Command
{
  public:
    virtual void execute() { jump(); }
};

class FireCommand : public Command
{
  public:
    virtual void execute() { fireGun(); }
};

// You get the idea...
```

Input handler

In our input handler, we store a pointer to a command for each button:

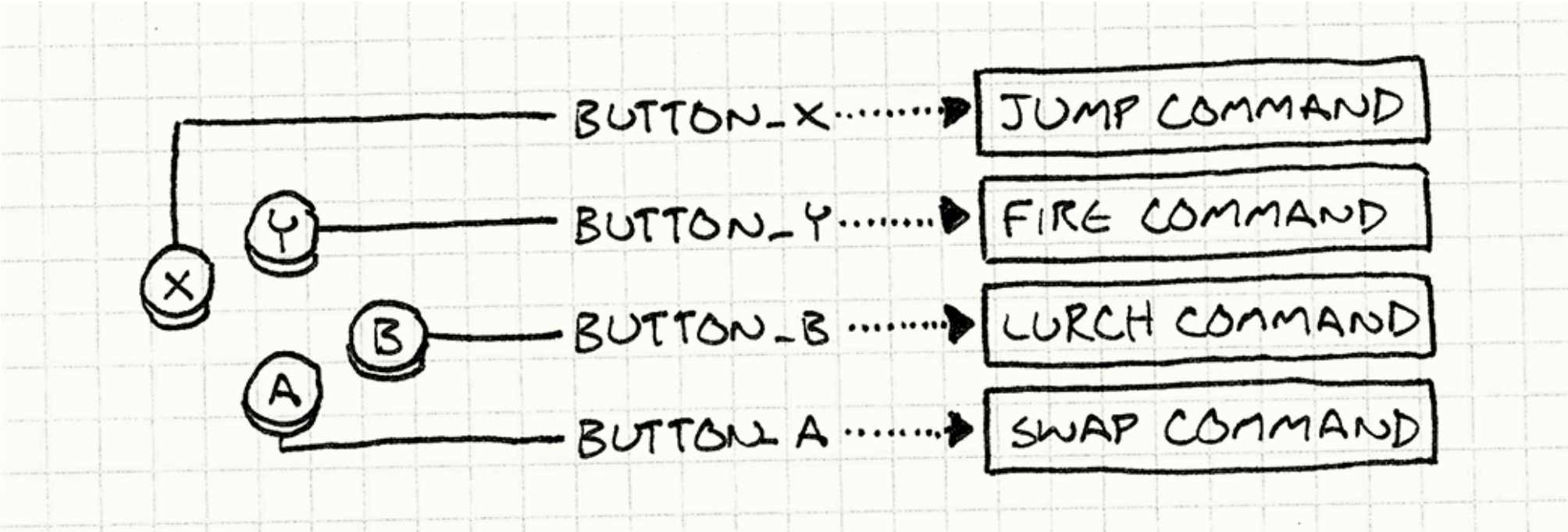
```
class InputHandler
{
  public:
    void handleInput();

    // Methods to bind commands...

  private:
    Command* buttonX_;
    Command* buttonY_;
    Command* buttonA_;
    Command* buttonB_;
};
```

Input handling

Now the input handling just delegates to those:

```
void InputHandler::handleInput()
{
  if (isPressed(BUTTON_X)) buttonX_->execute();
  else if (isPressed(BUTTON_Y)) buttonY_->execute();
  else if (isPressed(BUTTON_A)) buttonA_->execute();
  else if (isPressed(BUTTON_B)) buttonB_->execute();
}
```

Where each input used to directly call a function, now there's a layer of indirection:



Actors and Commands

- Why restrict ourselves to a single actor?

```
class Command
{
  public:
    virtual ~Command() {}
    virtual void execute(GameActor& actor) = 0;
};
```

- Now we can send an actor to the command
- `GameActor` is our “game object” class that represents a character in the game world
- We rewrite the commands:

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

HandleInput, again...

- We change `handleInput` to return the appropriate command object, rather than execute the action

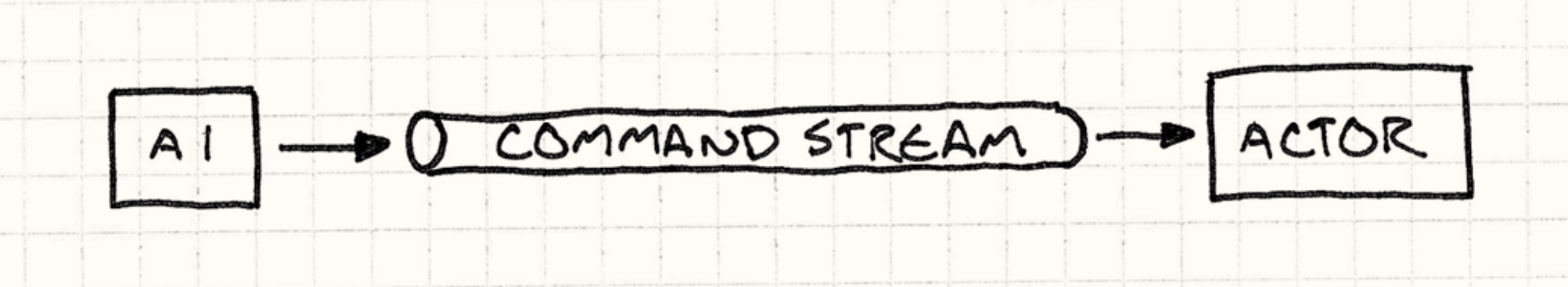
```
Command* InputHandler::handleInput()
{
  if (isPressed(BUTTON_X)) return buttonX_;
  if (isPressed(BUTTON_Y)) return buttonY_;
  if (isPressed(BUTTON_A)) return buttonA_;
  if (isPressed(BUTTON_B)) return buttonB_;

  // Nothing pressed, so do nothing.
  return NULL;
}
```

- We take advantage of the fact that the command is a reified call
  - We can delay the execution of the call
- Somewhere in the main loop:

```
Command* command = inputHandler.handleInput();
if (command)
{
  command->execute(actor);
}
```

So far…

- We can let the player control any actor in the game now by changing the actor we execute the commands on.
- This also allows us a neat AI implementation
  - AI simply emits command objects for each NPC actor



- This is the *decoupling* we were talking about. The command stream separates AI and the Actor.
- We can reassign actions to buttons if requested by the player

Undo and Redo

- If a command object can *do* things, it can also *undo* them
- Assume a move command for a unit:

```
class MoveUnitCommand : public Command
{
  public:
    MoveUnitCommand(Unit* unit, int x, int y)
    : unit_(unit),
      x_(x),
      y_(y)
    {}

    virtual void execute()
    {
      unit_->moveTo(x_, y_);
    }

  private:
    Unit* unit_;
    int x_, y_;
};
```

- Note that unlike the previous example, we now want to bind the actor to the action
- This is not a generic move command

Input handling

- Henceforth, our input handler must produce new commands for each new action-unit pair:

```
Command* handleInput()
{
  Unit* unit = getSelectedUnit();

  if (isPressed(BUTTON_UP)) {
    // Move the unit up one.
    int destY = unit->y() - 1;
    return new MoveUnitCommand(unit, unit->x(), destY);
  }

  if (isPressed(BUTTON_DOWN)) {
    // Move the unit down one.
    int destY = unit->y() + 1;
    return new MoveUnitCommand(unit, unit->x(), destY);
  }

  // Other moves...

  return NULL;
}
```

- Note that the commands are not executed at this stage

Undoable commands

- To make the commands undoable, we do the following modifications:

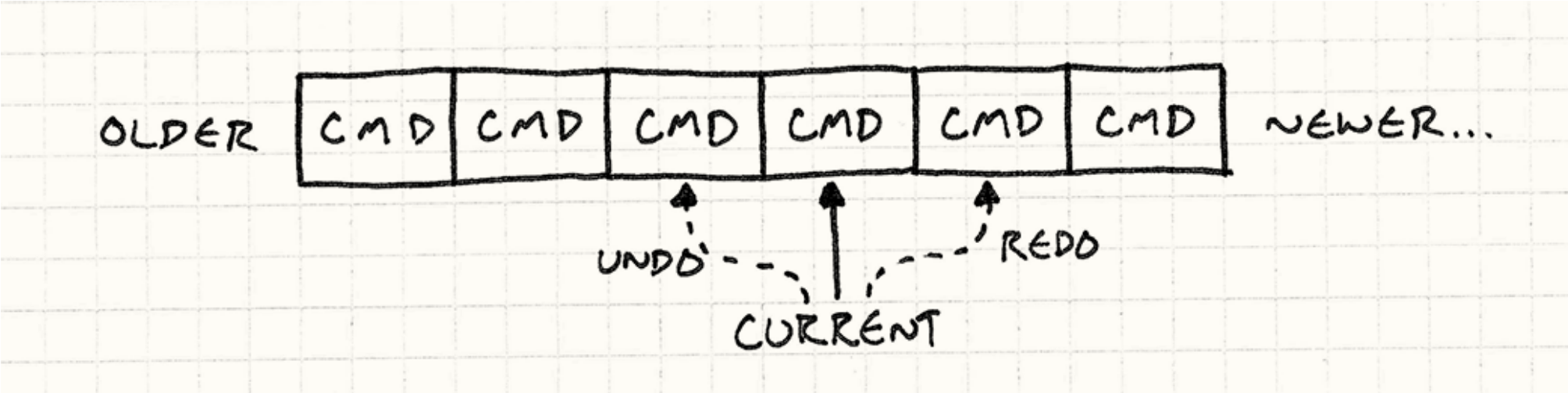
```
class Command
{
  public:
    virtual ~Command() {}
    virtual void execute() = 0;
    virtual void undo() = 0;
};
```

Undoable commands

An `undo()` method reverses the game state changed by the corresponding `execute()` method.

```
class MoveUnitCommand : public Command
{
  public:
    MoveUnitCommand(Unit* unit, int x, int y)
    : unit_(unit),
      xBefore_(0),
      yBefore_(0),
      x_(x),
      y_(y)
    {}

    virtual void execute() {
      // Remember the unit's position before the move, so we can restore it.
      xBefore_ = unit_->x();
      yBefore_ = unit_->y();
      unit_->moveTo(x_, y_);
    }

    virtual void undo() {
      unit_->moveTo(xBefore_, yBefore_);
    }

  private:
    Unit* unit_;
    int xBefore_, yBefore_;
    int x_, y_;
};
```

Undoable commands



- Supporting multiple levels of undo isn't much harder.
- Instead of remembering the last command, we keep a list of commands and a reference to the "current" one.
- When the player executes a command, we append it to the list and point "current" at it.

Flyweight Pattern

Flyweight Pattern

The fog lifts, revealing a majestic old growth forest. Ancient hemlocks, countless in number, tower over you forming a cathedral of greenery. The stained glass canopy of leaves fragments the sunlight into golden shafts of mist. Between giant trunks, you can make out the massive forest receding into the distance.

Forests of Polygons

- A sprawling woodland can be described with just a few sentences
- But actually implementing it in a realtime game is another story.
- When you've got an entire forest of individual trees filling the screen, all that a graphics programmer sees is the millions of polygons they'll have to somehow shovel onto the GPU every sixtieth of a second.
- 100 trees on screen
    - 1000 polygons each
        - 100000 polygons * 60 frames
            - 6000000 polygons to be rendered each second
- That must travel from the cpu to the gpu every second

Tree

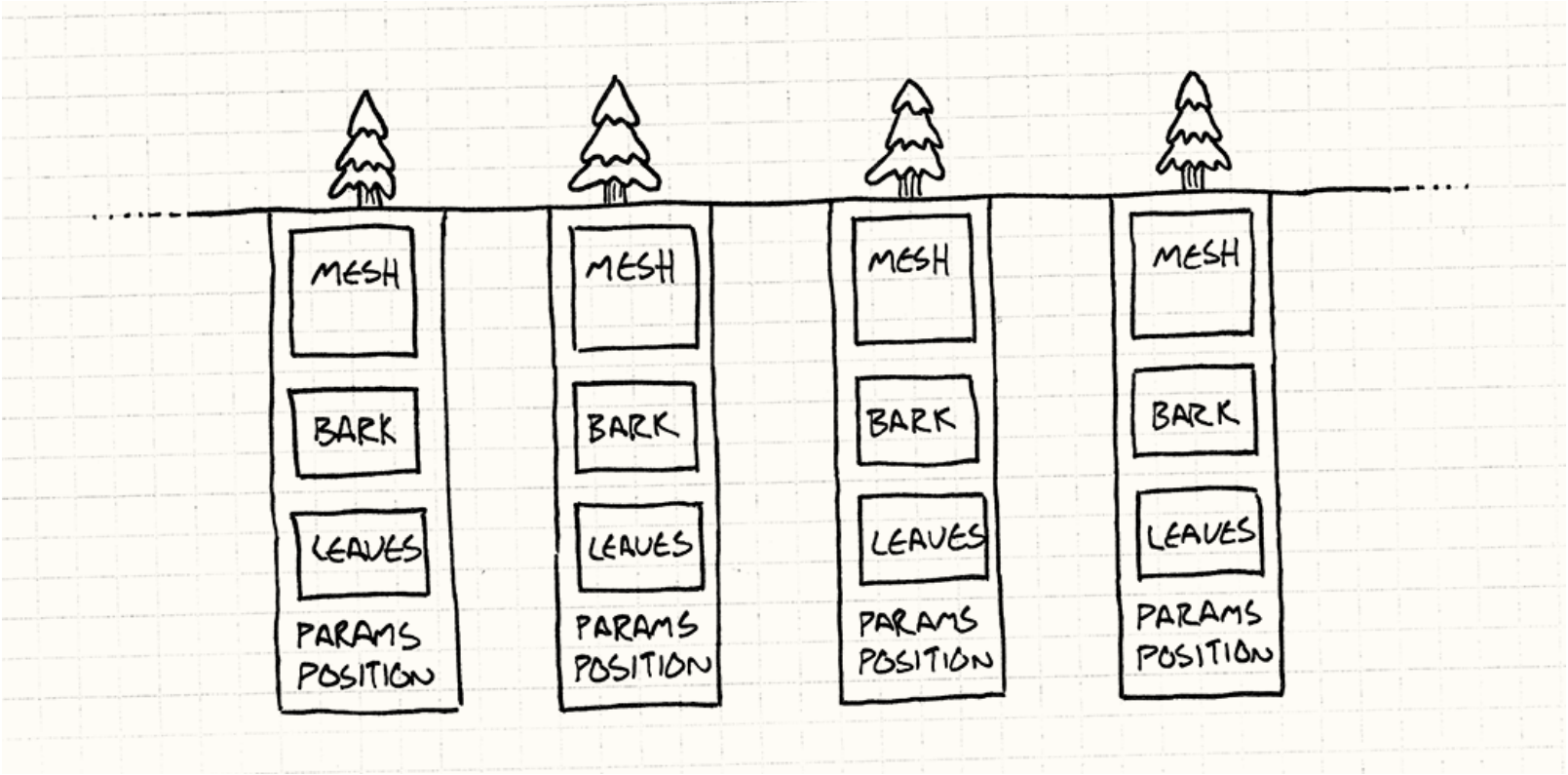
Each tree has a bunch of bits associated with it:

- A mesh of polygons that define the shape of the trunk, branches, and greenery.
- Textures for the bark and leaves.
- Its location and orientation in the forest.
- Tuning parameters like size and tint so that each tree looks different.

If you were to sketch it out in code, you'd have something like this:

```
class Tree
{
  private:
    Mesh mesh_;        // large data
    Texture bark_;     // large data
    Texture leaves_;   // large data
    Vector position_;
    double height_;
    double thickness_;
    Color barkTint_;
    Color leafTint_;
};
```

Tree



- Even though there may be thousands of trees in the forest, they mostly look similar.
- They can use the same mesh and textures.
- That means most of the fields in these objects are the same between all of those instances.

Skinny Trees

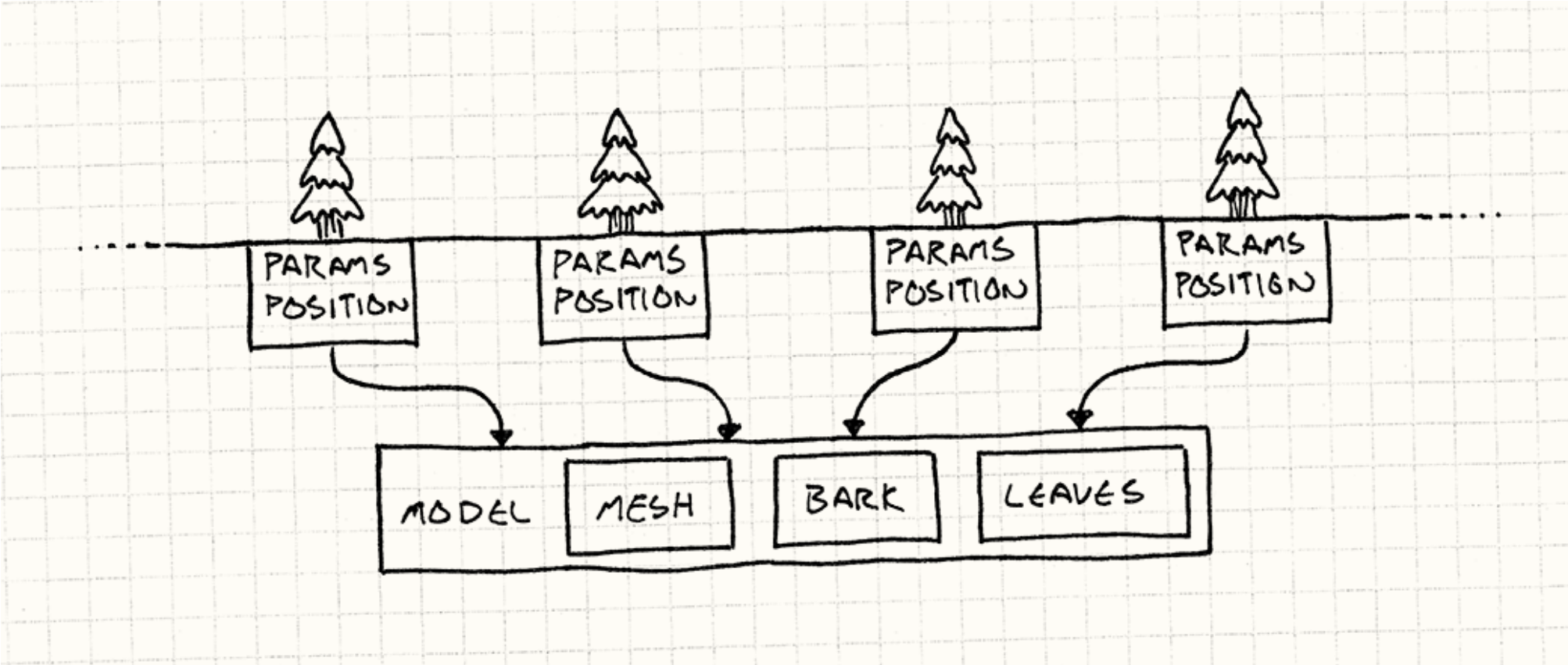
We can model that explicitly by splitting the object in half.

```
class TreeModel
{
  private:
    Mesh mesh_;
    Texture bark_;
    Texture leaves_;
};
```

The game only needs a single one of these, since there's no reason to have the same meshes and textures in memory a thousand times.

```
class Tree
{
  private:
    TreeModel* model_;

    Vector position_;
    double height_;
    double thickness_;
    Color barkTint_;
    Color leafTint_;
};
```

Skinny Trees



How to implement

- We have to send the shared data just once.
- Then, we send every tree instance's unique data
- Finally, we tell the GPU, "Use that one model to render each of these instances."
- Today's graphics APIs and cards support exactly that.
    - Both Direct3D and OpenGL can do something called *instanced rendering*.

Geometry Instancing

Starting in Direct3D version 9, Microsoft included support for geometry instancing. This method improves the potential runtime performance of rendering instanced geometry by explicitly allowing multiple copies of a mesh to be rendered sequentially by specifying the differentiating parameters for each in a separate stream. The same functionality is available in Vulkan core, and the OpenGL core in versions 3.1 and up but may be accessed in some earlier implementations using the EXT_draw_instanced extension.

A Place for Roots

- We need to have ground for these trees in our game
- The ground can be one of *grass*, *dirt*, *hills*, *river*, etc.
- We can simply use a *grid* to represent this variation
- Each cell of the grid can be one of these ground types
- Each terrain type has a number of properties that affect *gameplay*:
  - A *movement cost* that determines how quickly players can move through it.
  - A flag for whether it's a *watery terrain* that can be crossed by boats.
  - A *texture* used to render it.

A Place for Roots

A common approach is to use an *enum* for terrain types:

```
enum Terrain
{
  TERRAIN_GRASS,
  TERRAIN_HILL,
  TERRAIN_RIVER
  // Other terrains...
};
```

Then the world maintains a huge grid of those:

```
class World
{
  private:
    Terrain tiles_[WIDTH][HEIGHT];
};
```

Terrain Types

To actually get the useful data about a tile, we do something like:

```
int World::getMovementCost(int x, int y)
{
  switch (tiles_[x][y])
  {
    case TERRAIN_GRASS: return 1;
    case TERRAIN_HILL:  return 3;
    case TERRAIN_RIVER: return 2;
      // Other terrains...
  }
}

bool World::isWater(int x, int y)
{
  switch (tiles_[x][y])
  {
    case TERRAIN_GRASS: return false;
    case TERRAIN_HILL:  return false;
    case TERRAIN_RIVER: return true;
      // Other terrains...
  }
}
```

- This works, but it is ugly
- This should be *data* not code
- And the code is smeared across a bunch of functions

Terrain Class

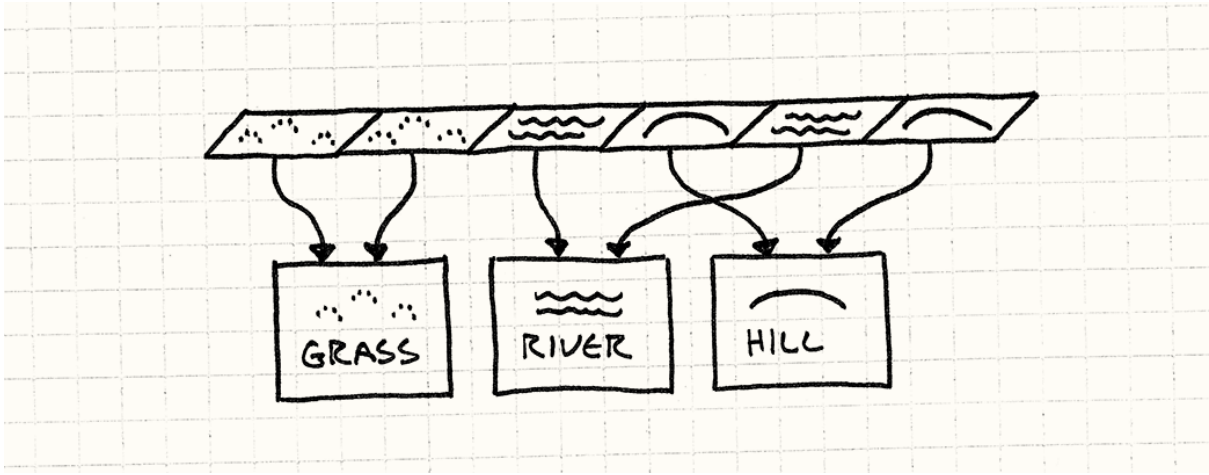
A terrain class can do the job for us:

```
class Terrain
{
public:
  Terrain(int movementCost,
          bool isWater,
          Texture texture)
  : movementCost_(movementCost),
    isWater_(isWater),
    texture_(texture)
  {}

  int getMovementCost() const { return movementCost_; }
  bool isWater() const { return isWater_; }
  const Texture& getTexture() const { return texture_; }

private:
  int movementCost_;
  bool isWater_;
  Texture texture_;
};
```

Notice that all of the methods here are const. Since the same object is used in multiple contexts, if you were to modify it, the changes would appear in multiple places simultaneously. That's probably not what you want. Because of this, Flyweight objects are almost always *immutable*.

Terrain class

- We have no instance based information in the Terrain class
- Having a separate Terrain object in each grid cell would be a huge waste




```
class World
{
  private:
    Terrain* tiles_[WIDTH][HEIGHT]; // pointers not objects

    // Other stuff...
};
```

World

Since the terrain instances are used in multiple places, their lifetimes would be a little more complex to manage if you were to dynamically allocate them. Instead, we'll just store them directly in the world:

```
class World
{
  public:
    World()
    : grassTerrain_(1, false, GRASS_TEXTURE),
      hillTerrain_(3, false, HILL_TEXTURE),
      riverTerrain_(2, true, RIVER_TEXTURE)
    {}

  private:
    Terrain grassTerrain_;
    Terrain hillTerrain_;
    Terrain riverTerrain_;

    // Other stuff...
};
```

Paint the world

Then we can use those to paint the ground like this:

```
void World::generateTerrain()
{
  // Fill the ground with grass.
  for (int x = 0; x < WIDTH; x++)
  {
    for (int y = 0; y < HEIGHT; y++)
    {
      // Sprinkle some hills.
      if (random(10) == 0)
      {
        tiles_[x][y] = &hillTerrain_;
      }
      else
      {
        tiles_[x][y] = &grassTerrain_;
      }
    }
  }

  // Lay a river.
  int x = random(WIDTH);
  for (int y = 0; y < HEIGHT; y++) {
    tiles_[x][y] = &riverTerrain_;
  }
}
```

Decoupling

Now instead of methods on World for accessing the terrain properties, we can expose the Terrain object directly:

```
const Terrain& World::getTile(int x, int y) const
{
  return *tiles_[x][y];
}
```

This way, World is no longer coupled to all sorts of details of terrains. If you want some property of the tile, you can get it right from that object:

```
int cost = world.getTile(2, 3).getMovementCost();
```