Ders 4

Oyun Programlama

Tasarım Örüntüleri

Command (Emir) Örüntüsü

Nedir?

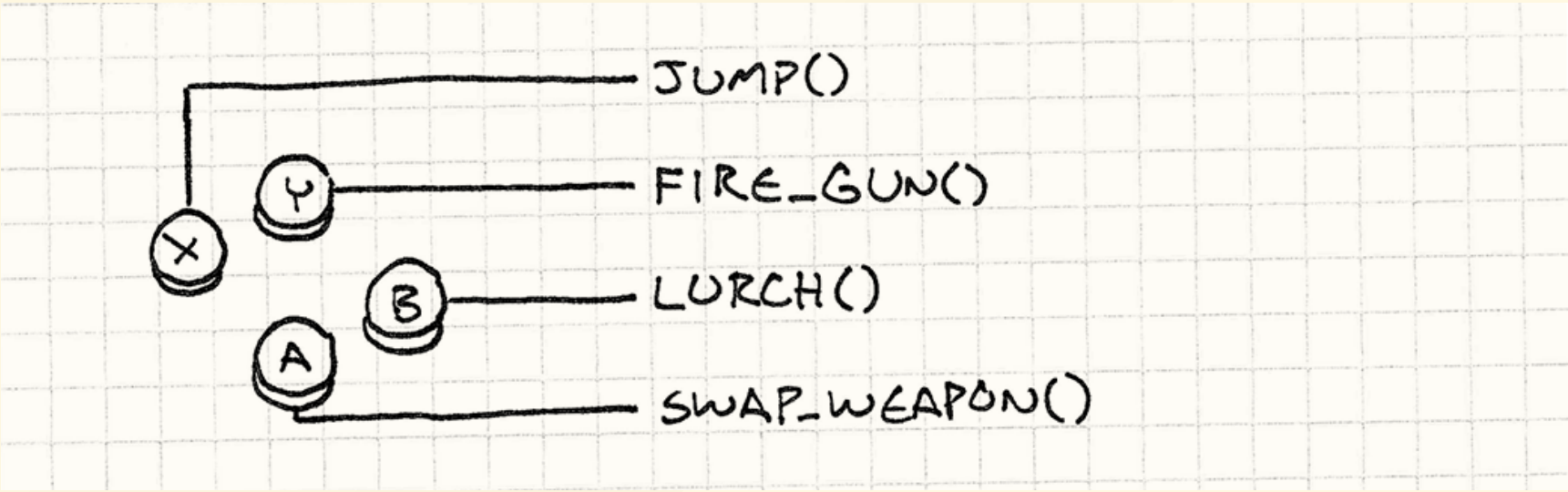
- Emir örüntüsü, bir metod çağrısının bir obje ile sarılmasıdır.
- Bu şekilde metod çağrısı daha kontrol edilebilir hale gelir.
- Mesela emirler bir sıraya konulabilir, geri alınabilir, vs.

Diğer isimleri

- Benzer örüntüler şu isimlerle de bilinir:
  - Callback
  - First-class function
  - Function pointer
  - Closure
  - Partially applied function

Örnek: Girdi Ayarları

Kullanıcı girdisi




```
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();
}
```

Kullanıcı girdisi

- Bu fonksiyon her framede `game_loop` içinde çağrılmaktadır.
- **Sorun**: düğmelerin görevini değiştirmek isteyen oyuncu
- Hardcode edilmiş olduğundan mümkün değil

```
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();
}
```

Base class

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

Her aksiyon için bir sub-class

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

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

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

Input handler

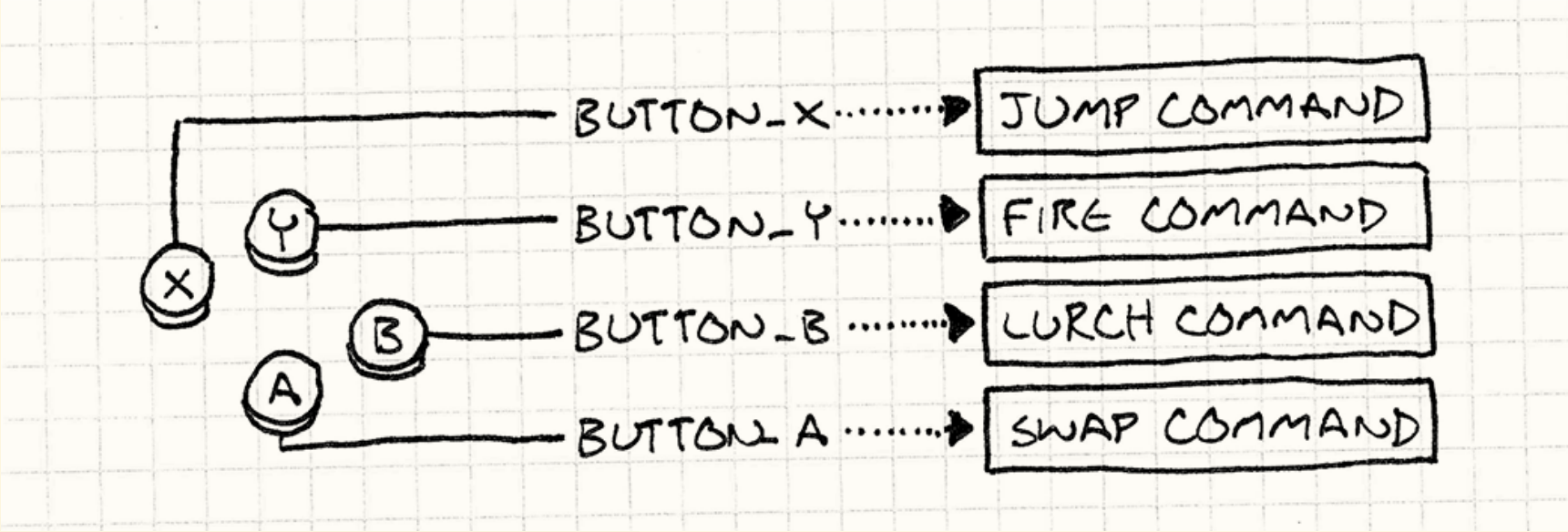
```
class InputHandler
{
  public:
    void handleInput();

    // Methods to bind commands...

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

Input handler ve command pointerları

```
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();
}
```

```
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();
}
```

What it looks like



Dahası: Aktör tabanlı komutlar

- Emir örüntüsü ile yalnızca avatarı değil, tüm aktörleri yönetebiliriz:

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

Aktör tabanlı komutlar

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

New `handleInput`

- Artık fonksiyonu çağırmak yerine bir emir dönüyor

```
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;
}
```

Hepsini birleştirmek

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

- `actor` ü değiştirerek, istediğimiz karakteri kontrol edebiliriz.

AI açısından bakış

- AI modülü `Command` üretir
- Bu `Command` objeleri uygun `actor` lere gönderilir
- AI modülleri, aktörler için plug edilecek objeler halini alır

Geri al / Tekrar et (Undo/Redo)

- En bilinen use-caselerden biridir
- Gerçek zamanlı oynanmayan bir çok oyunda geri alma özelliği vardır
- Aynı zamanda level editörleri vs. için de gereklidir

Örnek: Unit Movement

```
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_;
};
```

handleInput

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

  if (isPressed(BUTTON_UP)) {
    int destY = unit->y() - 1;
    return new MoveUnitCommand(unit, unit->x(), destY);
  }

  if (isPressed(BUTTON_DOWN)) {
    int destY = unit->y() + 1;
    return new MoveUnitCommand(unit, unit->x(), destY);
  }

// Other moves...
  return NULL;
}
```

`Command` classında değişiklik

- Yeni eklenen `undo()` metodu

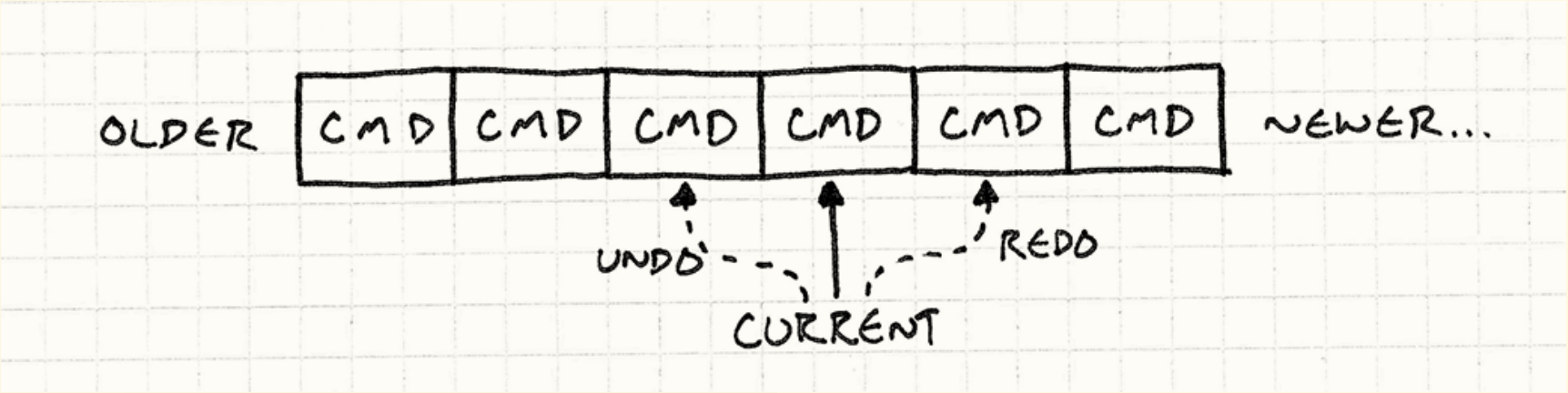
```
class Command
{
public:
  virtual ~Command() {}
  virtual void execute() = 0;
  virtual void undo() = 0;
};
```

`MoveUnitCommand` değişiklikleri

```
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_;
};
```

Emir yığını (Command stack)



- Oyun tekrar oynatmaları (game replay) için faydalı!

First-class fonksiyonlar

```
function makeMoveUnitCommand(unit, x, y) {
  var xBefore, yBefore;
  return {
    execute: function() {
      xBefore = unit.x();
      yBefore = unit.y();
      unit.moveTo(x, y);
    },
    undo: function() {
      unit.moveTo(xBefore, yBefore);
    }
  };
}
```

Sineksiklet (flyweight) Örüntüsü

Bir oyun sahnesi

“ 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. ”

“ Sis çözülür, ve görkemli yaşlı ormanı açığa çıkarır. Sayısız, kadim baldıran otu, üzerinizde bir yeşillik katedrali oluşturarak yükselir. Yaprakların vitraylı gölgeliği güneş ışığını altın sis şaftlarına böler. ”

Anlatmak kolay, yapmak zor

- Böyle bir sahne milyonlarca poligon demek
- Saniyede 60 defa güncellenecek ve tekrar çizilecek milyonlarca poligon

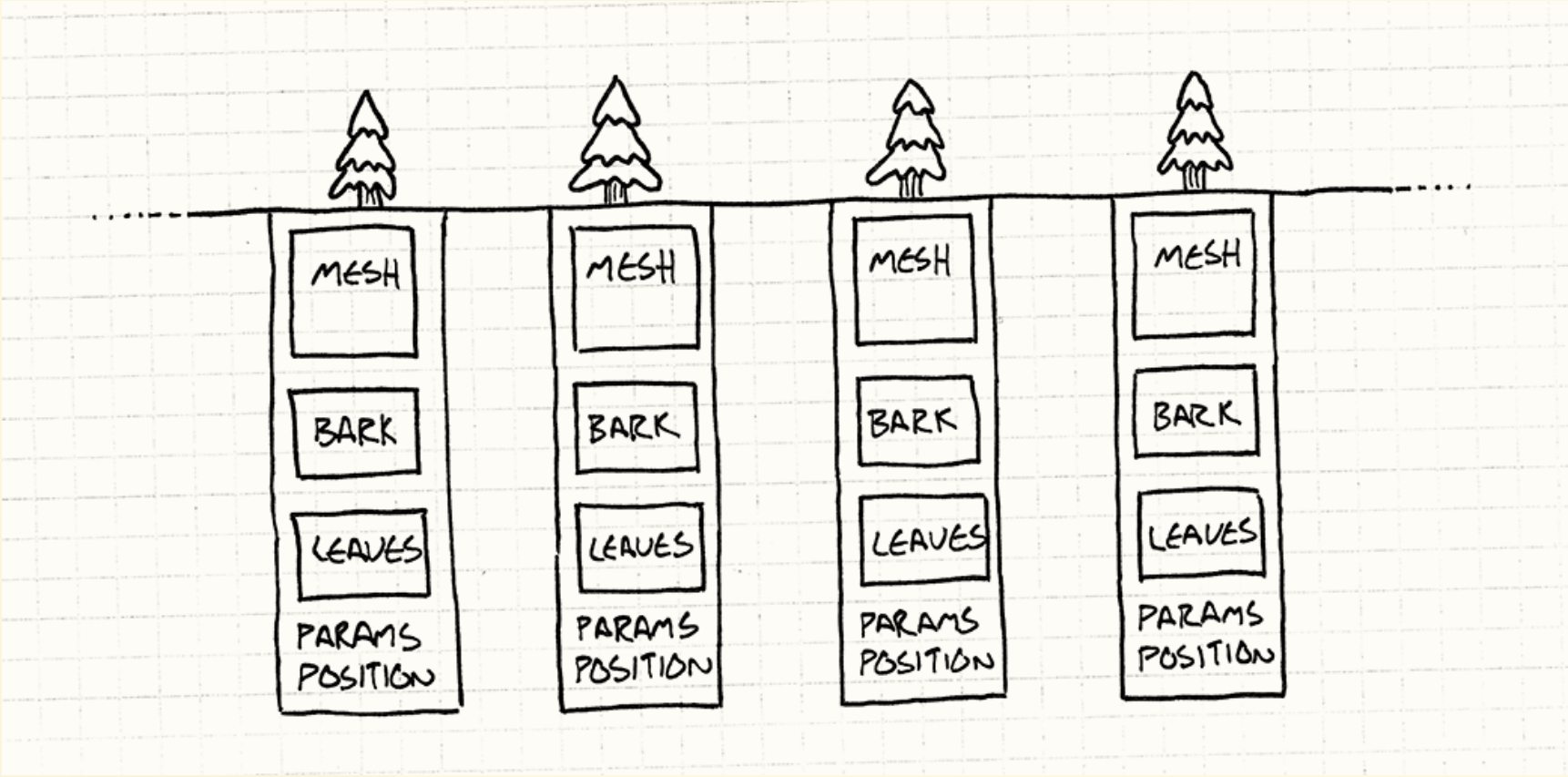
Her bir ağacın özellikleri

- Ağacın unsurlarını tanımlayan poligon mesh
- Texture (desen)
- Yer ve yön bilgisi
- Büyüklük ve tint bilgisi

Örnek kod : Tree

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

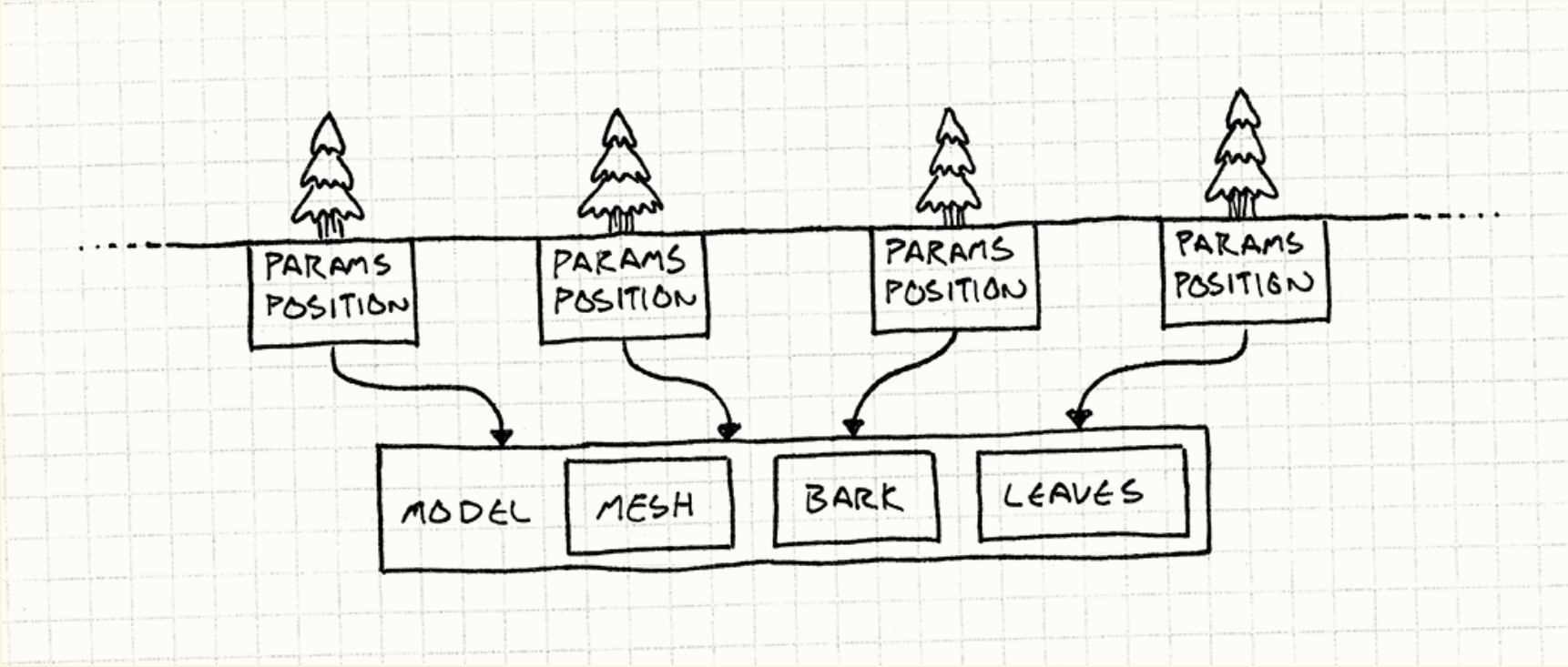
Ortak özelliklerin ayrıştırılması



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

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

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

Nasıl görünüyor?



> Modeli GPU'ya bir kerede gönderemezsek bir anlamı yok!

Instanced Rendering

- Hem OpenGL, hem de Direct3D destekliyor

Geometry instancing

From Wikipedia, the free encyclopedia

In real-time computer graphics, **geometry instancing** is the practice of rendering multiple copies of the same mesh in a scene at once. This technique is primarily used for objects such as trees, grass, or buildings which can be represented as repeated geometry without appearing unduly repetitive, but may also be used for characters. Although vertex data is duplicated across all instanced meshes, each instance may have other differentiating parameters (such as color, or skeletal animation pose) changed in order to reduce the appearance of repetition.

Contents

API support for geometry instancing

Starting in Direct3D version 9, Microsoft included support for geometry instancing. This method improves the potential runtime

Peki sineksiklet örüntüsü nedir?

- Eğer bir nesneden çok fazla varsa, onun hafif olması iyidir
- Her nesneyi iki parçaya ayırmaya çalışın
  - Tüm instance'larda ortak olan kısım
  - Her instance'a özel olan kısım
- Peki amaç sadece hafızadan tasarruf mu?

Bir başka örnek: Zemin

- Bu ağaçların yerleştirileceği zemini düşünelim
- Tile-based olsun
  - Her tile'ın bir türü olsun (çimen, tepe, nehir)
  - Her tile türünde oyuncunun hızı farklılaşsın
  - İçinde su olup olmadığı (tekne kullanımı)
  - Desen (texture)

`Terrain` enumeratörü

- Tüm bilgileri her tile'da ayrıca tutmak istemeyiz
- Bunun yerine tile'in türünü bilmek yeterlidir

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

World

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

Detaylara erişim

```
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;
  }
}

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;
  }
}
```

Dağınıklık

- Bu şekilde fazlaca dağınık oldu
- Bir zemin türünün bilgisi farklı farklı yerlerde tutuluyor

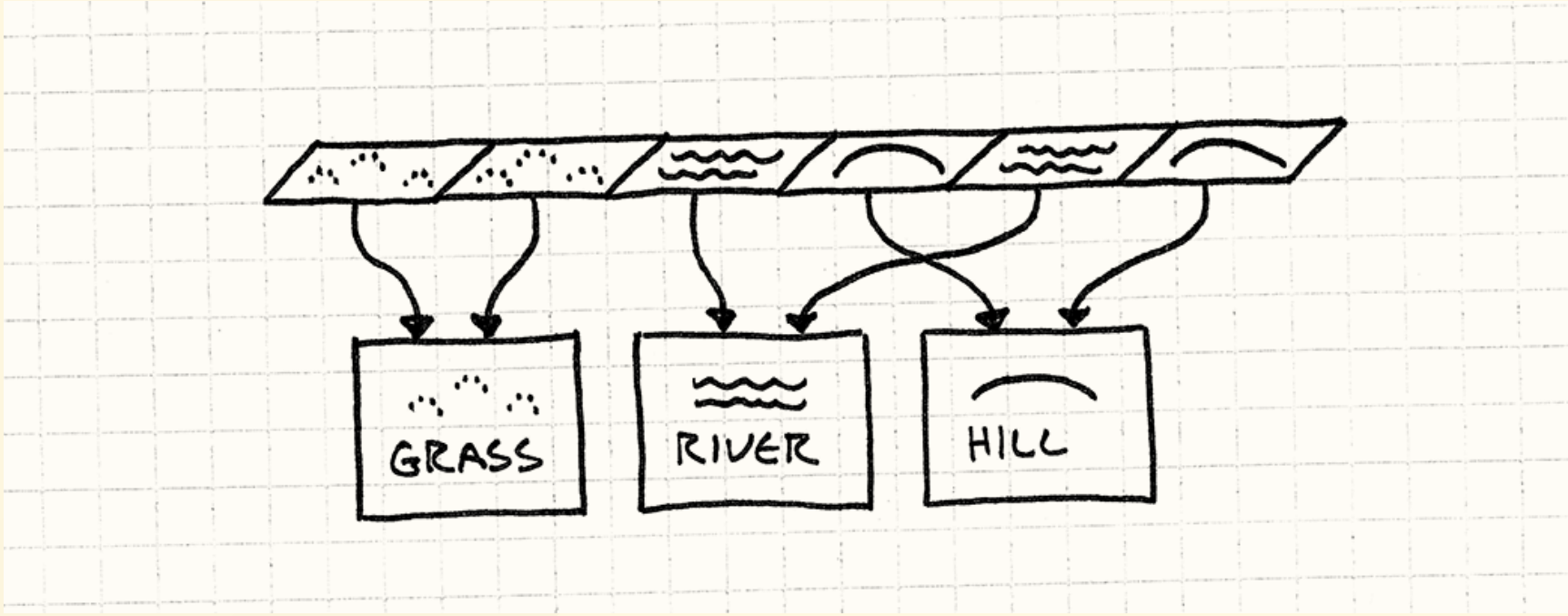
Daha topluca : `Terrain` sınıfı

```
class Terrain {
  public:
    Terrain(int moveCost, bool isWater, Texture texture)
    : movementCost_(moveCost),
      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_;
};
```

World

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

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

Nasıl görünüyor



`World`, `Terrain` nesnelerini de tutabilir

```
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...
};
```

Örnek: Zeminin hazırlanması

```
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_;
  }
}
```

Erişim

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

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

Prototip Örüntüsü

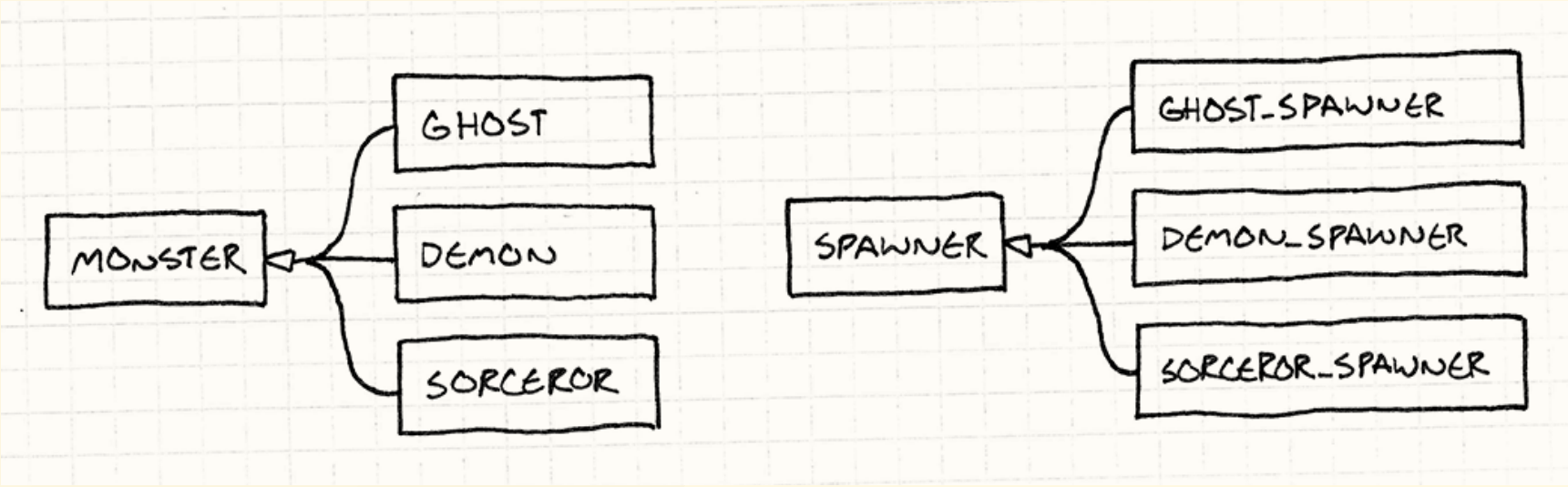
Nedir?

- *Gauntlet* türü bir oyun yaptığımızı farzedin

Nedir?

```
class Monster
{
  // Stuff...
};

class Ghost : public Monster {};
class Demon : public Monster {};
class Sorcerer : public Monster {};
```

Spawners


MONSTER
GHOST
DEMON
SORCEROR
SPAWNER
GHOST_SPAWNER
DEMON_SPAWNER
SORCEROR_SPAWNER

```
class Spawner {
  public:
    virtual ~Spawner() {}
    virtual Monster* spawnMonster() = 0;
};

class GhostSpawner : public Spawner {
  public:
    virtual Monster* spawnMonster() {
      return new Ghost();
    }
};

class DemonSpawner : public Spawner {
  public:
    virtual Monster* spawnMonster() {
      return new Demon();
    }
};
```

Sorunlar

- Çok fazla sınıf
- Çok fazla tekrar
- Gereksiz kod

Prototipleme fikri

> Eğer elinizde bir nesne varsa, bu nesne kendisine benzer nesneler üretebilir.

- Her nesne benzerleri için bir prototiptir

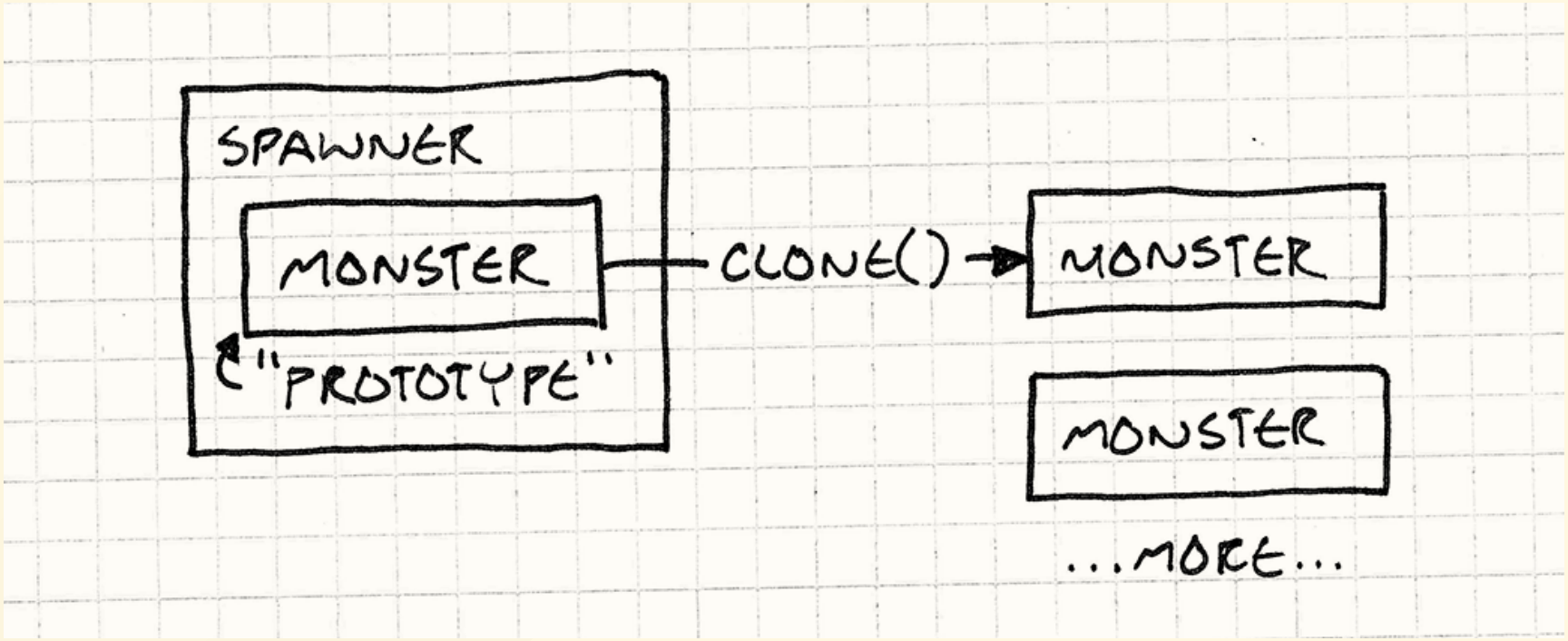
```
class Monster
{
  public:
    virtual ~Monster() {}
    virtual Monster* clone() = 0;

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

```
class Ghost : public Monster {
  public:
    Ghost(int health, int speed)
    : health_(health),
      speed_(speed)
    {}

    virtual Monster* clone()
    {
      return new Ghost(health_, speed_);
    }

  private:
    int health_;
    int speed_;
};
```

Spawner kodu

```
class Spawner
{
  public:
    Spawner(Monster* prototype)
    : prototype_(prototype)
    {}

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

  private:
    Monster* prototype_;
};
```

Nasıl görünüyor?



Sıkıntılar

- Hala bir sürü clone metodu yazılması gerekiyor
- Depp clone vs. Shallow clone
- Clone by nature is a difficult process

Spawn fonksiyonları

```
Monster* spawnGhost() { return new Ghost(); }

typedef Monster* (*SpawnCallback)();

class Spawner {
  public:
    Spawner(SpawnCallback spawn) : spawn_(spawn)
    {}

    Monster* spawnMonster() {
      return spawn_();
    }

  private:
    SpawnCallback spawn_;
};
```

Template (Şablon) kullanımı

```
class Spawner
{
  public:
    virtual ~Spawner() {}
    virtual Monster* spawnMonster() = 0;
};

template <class T>
class SpawnerFor : public Spawner
{
  public:
    virtual Monster* spawnMonster() { return new T(); }
};
```

```
Spawner* ghostSpawner = new SpawnerFor<Ghost>();
```