Приспособленец (Flyweight)

**1. Лес для деревьев – как мы хотим описать лес, в чем проблема, какие данные связаны с каждым деревом, в чем фишка их данных, позволяющих применить шаблон Приспособленец, как мы делим данные у нашего объекта (итоговый код), как это можно себе визуализировать**

**2. Тысяча экземпляров – API современных видеокарт для рендеринга (как называется), какие два потока формирует**

**3. Шаблон приспособленец (теоретическая часть) – когда его используем, в чем большая проблема – в необходимом количестве памяти или времени, которое нужно, чтобы эти данные прогнать через шину видеокарты, как шаблон решает проблему, как это описывает банда четырех**

**4. Место, где можно пустить корни – как мы представим землю в нашей игре, ее параметры, первый вариант реализации, как это сделать через наш шаблон Приспособленец (второй вариант), как мы до этого хранили сетку нашу, как теперь, как это можно визуализировать, как храним теперь объекты наших плиток**

Я могу описать целый лес всего несколькими предложениями, но реализация его в настоящей игре — совсем другая история. Если вам захочется вывести на экране весь лес из индивидуальных деревьев целиком, любой графический программист сразу увидит миллионы полигонов, которые придется обработать видеокарте на каждом кадре.

Мы говорим именно о тысячах деревьев, геометрия каждого из которых достаточно детализирована и насчитывает тысячи полигонов. Даже, если у вас найдется достаточно памяти, чтобы это все уместить, для того, чтобы отрендерить лес целиком, вам нужно будет пропустить это все через шины процессора и видеокарты.

С каждым деревом связаны следующие данные:

* Полигональная сетка, описывающая его ствол, ветви и листву.
* Текстура коры и листьев.
* Положение и ориентация в лесу.
* Индивидуальные настройки, такие как размер и оттенок, благодаря которым каждое дерево в лесу будет выглядеть индивидуально.

Если набросать описывающий все это код, получится нечто подобное:

class Tree

{

private:

Mesh mesh\_;

Texture bark\_;

Texture leaves\_;

Vector position\_;

double height\_;

double thickness\_;

Color barkTint\_;

Color leafTint\_;

};

Целая куча данных и здоровенные полигональная сетка и текстура. Весь лес определенно не получится запихнуть целиком в видеокарту на каждом кадре. К счастью есть проверенный временем способ обойти это ограничение. Дело тут в том, что даже несмотря на то, что деревьев в лесу тысячи, выглядят они довольно похоже. Потому что все используют одни и те же сетку и текстуру. Это значит, что большинство полей в объекта идентичны для всех его экземпляров.



Обратите внимание, что данные в прямоугольнике одинаковы для всех деревьев.

Мы можем это явно cмоделировать путем разделения объекта пополам. Мы вытаскиваем общие для всех деревьев данные, и перемещаем их в отдельный класс:

class TreeModel

{

private:

Mesh mesh\_;

Texture bark\_;

Texture leaves\_;

};

Игре нужен только один экземпляр этого класса, потому что нам незачем иметь тысячи копий одной и той же сетки и текстуры. Поэтому каждый экземпляр дерева в мире должен иметь **ссылку** на общий TreeModel. Остальные данные являются индивидуальными для каждого экземпляра:

class Tree

{

private:

TreeModel\* model\_;

Vector position\_;

double height\_;

double thickness\_;

Color barkTint\_;

Color leafTint\_;

};

Результат можно изобразить следующим образом:



Это конечно все хорошо и экономит нам кучу памяти, но как это поможет нам в рендеринге? Прежде чем лес появится на экране, его нужно пропустить через память видеокарты. Нам нужно организовать наше разделение ресурсов таким образом, чтобы это было понятно видеокарте.

**2. Тысяча экземпляров**

Для того чтобы минимизировать количество данных, передаваемых видеокарте, мы будем передавать общие данные, т.е. TreeModel только один раз. После этого мы будем по отдельности передавать индивидуальные для каждого экземпляра данные — позицию, цвет и размер. А затем просто скомандуем видеокарте "Используй эту модель для отрисовки всех этих экземпляров".

К счастью, API современных видеокарт такую возможность поддерживает. Детали конечно гораздо сложнее и выходят за рамки рассмотрения этой книги, однако и в Direct3D и в OpenGL присутствует возможность рендеринга экземпляров (instanced rendering).

В обеих API вы формируете два потока данных. Первый — это общие данные, которые используются много раз — сетки и текстуры, как в нашем примере. Второй — список экземпляров и их параметры, которые позволяют варьировать данные из первой группы данных во время отрисовки. Весь лес появляется после единственного вызова отрисовки.

3. Шаблон Приспособленец

Теперь, когда у нас есть хороший пример, я могу рассказать вам о самом шаблоне. Приспособленец, как следует из его имени, вступает в игру, когда нам требуется **максимально облегченный объект**, обычно потому что нам нужно очень много таких объектов.

При использовании метода рендеринга экземпляров (instanced rendering) дело даже не в том, что нужно много памяти, а в том, что требуется **слишком много времени**, чтобы прокачать данные о каждом дереве через шину видеокарты. Главное, что базовая идея общая.

Шаблон решает эту проблемы с помощью разделения данных объекта на два типа: **первый тип данных — это неуникальные для каждого экземпляра объекта данные, которые можно иметь в одном экземпляре для всех объектов**. Банда четырех называет их **внутренним (intrinsic) состоянием**, но мне больше нравится название "контекстно-независимые". В нашем примере это геометрия и текстура дерева.

**Оставшиеся данные — это внешнее (extrinsic) состояние, все что уникально для каждого экземпляра**. В нашем случае это позиция, масштаб и цвет каждого из деревьев. Также как и в приведенном выше фрагменте кода, этот шаблон предотвращает перерасход памяти с помощью разделения одной копии внутреннего состояния между всеми местами, где оно появляется.

То, что мы до сих пор видели, выглядит как простое разделение ресурсов и вряд ли заслуживает того, чтобы называться шаблоном. Частично это вызвано тем, что в нашем примере присутствует очевидное свойство (identity), выделяемое в качестве разделяемого ресурса — TreeModel.

Я считаю этот шаблон менее очевидным (и поэтому более хитрым), когда он используется в случае, где выделить свойства для разделяемого объекта не так легко. В таком случае возникает впечатление, что объект магическим образом оказывается в нескольких местах одновременно. Давайте я продемонстрирую это на примере.

4. Место, где можно пустить корни

Земля, на которой растут наши деревья, тоже должна быть представлена в игре. Это могут быть участки травы, грязи, холмов, озер и рек и любой другой типа местности, который вы только сможете придумать. Мы сделаем нашу землю на основе тайлов (tile - плитка): вся поверхность будет состоять из отдельных маленьких плиток. Каждая плитка будет относиться к какому-либо типу местности.

Каждый из типов местности имеет ряд параметров, влияющих на геймплей:

* Стоимость перемещения, определяющая скорость с которой игроки могут по ней двигаться.
* Флаг, означающий что местность залита водой и по ней можно перемещаться на лодке.
* Используемая для рендеринга текстура.

Так как все мы — игровые программисты параноики в плане эффективности, мы точно не станем хранить все эти данные для каждого тайла в игровом мире. Вместо этого обычно используется перечисление для описания типов местности:

enum Terrain

{

TERRAIN\_GRASS,

TERRAIN\_HILL,

TERRAIN\_RIVER

// Other terrains...

};

А сам мир хранит здоровенный массив (сетку) этих значений:

class World

{

private:

Terrain tiles\_[WIDTH][HEIGHT];

};

Чтобы получить полезную информацию о тайле, используется нечто наподобие:

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...

}

}

Ну вы поняли идею. Такой подход работает, но я нахожу его уродливым. Когда я думаю о скорости перемещения по местности, я предполагаю увидеть данные о местности, а они вместо этого внедрены в код. Еще хуже то, что данные об одном типе местности размазаны по целой куче методов. Хотелось бы видеть все это инкапсулированным в одном месте. В конце концов именно для этого и предназначены объекты.

Было бы здорово иметь настоящий класс для местности наподобие такого:

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\_;

};

Вы можете заметить, что все методы здесь объявлены как **const**. Это не совпадение. Так как один и тот же объект используется в различном контексте, если мы его изменим, изменения одновременно произойдут и во всех остальных местах, а это не то, что вам нужно. Разделение объектов в целях экономии памяти — это оптимизация, не влияющая на видимое поведение приложения. Вот поэтому объект Приспособленец практически всегда делают неизменным (immutable).

Но мы не можем согласиться с тем, что нам придется платить производительностью за роскошь иметь экземпляр данного класса для каждого тайла. Если вы внимательно посмотрите на класс, вы обратите внимание на то, что здесь **нет вообще никакой специфической информации**, определяющей где находится тайл. Т.е. в терминах приспособленца, все состояние местности является внутренним (intrinsic) или "контекстно-независимым".

Принимая это во внимание, у нас нет причин иметь больше одного объекта для каждого типа местности. Каждый тайл травы идентичен любому другому. Вместо того, чтобы иметь массив перечислений или объектов Terrain , мы будем хранить массив **указателей** на объекты Terrain:

class World

{

private:

Terrain\* tiles\_[WIDTH][HEIGHT];

// Other stuff...

};

Каждый из тайлов, относящийся к одному типу местности, указывает на один и тот же экземпляр местности.



Так как экземпляры местности используется в нескольких местах, управлять их временем жизни будет сложнее, чем если бы создавали их динамически. Вместо этого мы будем прямо хранить их в мире:

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...

};

Теперь мы можем использовать их для отрисовки земли следующим образом:

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\_;

}

}

Дальше, вместо методов в World для доступа к параметрам местности, мы можем просто возвращать объект Terrain напрямую:

const Terrain& World::getTile(int x, int y) const

{

return \*tiles\_[x][y];

}

Таким образом World больше не перегружен различной информацией о местности. Если вам нужны некоторые параметры тайла, вы можете получить их у самого объекта:

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

Мы снова вернулись к приятному API работы с реальным объектом и добились этого практически без дополнительных накладных расходов: указатель обычно занимает не больше места чем перечисление.