Метод обновления (Update Method)

1. Задача

2. Мотивация – пример с со скелетами и статуями, во что превращается код, в чем заключается сущность шаблона, который мы собираемся использовать, что нужно для этого сделать, когда это вызывать

3. Шаблон – описание (3 пункта)

4. Когда использовать – 3 пункта (как даже в настольной игре может пригодиться данный шаблон)

5. Имейте в виду – предостережение касательно данного шаблона

5.1. Разделение кода на срезы отдельных кадров делает его сложнее – сравнение банального передвижения скелетов и внедрения этого кода в игровой цикл (их сложности)

5.2. Вам нужно сохранять состояние, чтобы продолжать с того места, где вы прервались – какую переменную нам пришлось добавить для перехода в кадровый режим, какой шаблон здесь может помочь

5.3. Объекты симулируются на каждом кадре, но не в настоящем конкурентном режиме – что под этим понимается (как у нас работает обновление объектов и какая проблема может возникнуть), пример с обновлением А и В, какой нам шаблон здесь поможет, однако всегда ли игры устроены так (пример с шахматами), как решить проблему для шахмат?

5.4. Будьте осторожны при изменении списка объектов во время обновления – что обычно относится к коду обновления объектов, пример с дропом предмета после смерти скелета, как решить эту проблему, когда проблема усложняется и что может произойти в таком случае, простейшее решение этой проблемы, как можно еще решить эту проблему

6. Пример кода – класс сущности (код), класс мира (код), реализация метода игрового цикла у класса мира (код)

7. Подклассы сущностей?! – что мы сразу представляем после определения базового класса сущности, но в чем проблема, что про это говорили Банда Четырех, что говорит автор, когда это понимание проникло в игровую индустрию, какой появился шаблон, что происходит с методом обновления, когда мы его применяем

8. Определение сущностей – реализация скелета и статуи через наследование от базовой сущности, какую особенность можно подметить, истинная цель шаблона

9. Ход времени – ранее мы предполагали, что каждый вызов метода обновления продвигает состояние игрового мира на \_ отрезок времени, однако многие игры используют \_ \_ шаг, что нам нужно изменить в связи с этим в методе обновления? Пример на методе скелетов

10. Архитектурные решения

10.1. В каком классе будет жить метод обновления? – три варианта (приведенные автором), подробно о каждом

10.2. Как обрабатываются спящие объекты? – какие специальные объекты присутствуют в играх, в чем может быть расточительность? Какой выход из этой ситуации? Минусы двух подходов к хранению объектов

**1. Задача**

Симуляция коллекции независимых объектов с помощью указания каждому объекту обработки одного кадра поведения за раз.

**2. Мотивация**

Могучая валькирия игрока выполняет квест по краже прекрасных украшений с трупа давно умершего короля-волшебника. Она приближается ко входу величественной усыпальницы и ее атакует... **ничего**. Никаких проклятых статуй, стреляющих молниями. Никаких воинов нежити, патрулирующих вход. Она просто заходит. Забирает лут. Игра окончена. Вы выиграли.

Ну нет. Так не пойдет.

Гробнице нужны стражи — противники, с которыми сможет побороться наша героиня. Для начала нам понадобятся ожившие скелеты воины, патрулирующие вход. Если вы проигнорируете все, что знаете об игровом программировании, простейший код, перемещающий скелетов туда и сюда будет выглядеть так:

while (true)

{

// Patrol right.

for (double x = 0; x < 100; x++)

{

skeleton.setX(x);

}

// Patrol left.

for (double x = 100; x > 0; x--)

{

skeleton.setX(x);

}

}

Проблема здесь в том, что хотя скелеты и двигаются туда-сюда, но игрок их не видит. Программа зациклена в бесконечном цикле и никакого игрового процесса тут нет. Чего мы на самом деле хотим добиться — так это того, чтобы **скелеты двигались на каждом кадре**.

Уберем эти циклы и переложим работу на уже существующий цикл. Это позволит игре реагировать на пользовательский ввод и рендерить врагов во время их перемещения. Вот так:

Entity skeleton;

bool patrollingLeft = false;

double x = 0;

// Main game loop:

while (true)

{

if (patrollingLeft)

{

x--;

if (x == 0) patrollingLeft = false;

}

else

{

x++;

if (x == 100) patrollingLeft = true;

}

skeleton.setX(x);

// Handle user input and render game...

}

Я привожу здесь код до и после, чтобы показать вам насколько код усложнился. Патрулирование влево и вправо может быть простым циклом for. Они даже косвенно следят за направлением скелетов в зависимости от запущенного цикла. Но теперь нам придется на каждом кадре прерывать выполнение и переходить в игровой цикл, а потом продолжать с того места, на котором мы остановились. А чтобы определять направление движения, нам придется использовать дополнительную переменную patrollingLeft.

Этот вариант уже более-менее рабочий. Безмозглый мешок с костями не составит вашей Северной воительнице серьезную конкуренцию, так что следующее, что мы добавим — это зачарованные статуи. Они будут стрелять молниями так часто, что героине придется прокрадываться мимо них на цыпочках.

Продолжая в нашем "максимально простом стиле", получим вот что:

// Skeleton variables...

Entity leftStatue;

Entity rightStatue;

int leftStatueFrames = 0;

int rightStatueFrames = 0;

// Main game loop:

while (true)

{

// Skeleton code...

if (++leftStatueFrames == 90)

{

leftStatueFrames = 0;

leftStatue.shootLightning();

}

if (++rightStatueFrames == 80)

{

rightStatueFrames = 0;

rightStatue.shootLightning();

}

// Handle user input and render game...

}

Не могу сказать что мы движемся в сторону кода, который легко и приятно поддерживать. У нас появилась куча новых переменных, а внутри игрового цикла образовалось месиво из кода, каждая часть которого обрабатывает отдельную сущность в игре. Чтобы они все работали одновременно, у нас получилось месиво относящегося к ним кода. Каждый раз когда ваш код можно описать словом "месиво" — у вас явно есть проблема.

Шаблон, который мы будем использовать для решения этой проблемы настолько прост, что вы наверное уже и сами до него додумались: **каждая сущность в игре инкапсулирует собственное поведение**. Таким образом наш игровой цикл становится лаконичным и мы получаем возможность обрабатывать любое количество сущностей.

Чтобы это сделать, нам нужен уровень абстракции и мы создаем его, определяя абстрактный метод update(). Игровой цикл поддерживает коллекцию объектов, но не знает их конкретный тип. Все что о них нужно знать — это то, что их нужно обновлять. Таким образом мы отделяем поведение каждого объекта от игрового цикла и других объектов.

На каждом кадре игровой цикл проходит по всей коллекции и вызывает update() для каждого объекта. Таким образом каждый объект может обновить свое поведение на один кадр. Вызывая его для объектов на каждом кадре, мы получаем их одновременное действие.

Так как мне обязательно кто-то это припомнит, сознаюсь сразу — да, они не работают в полностью конкурентном режиме. Пока один из объектов обновляется, все остальные этого не делают. Мы еще вернемся к этому позже.

Игровой цикл содержит динамическую коллекцию объектов, так что добавлять и удалять объекты с уровня довольно просто — просто добавляем или удаляем их из коллекции. Больше никакого хардкодинга. Более того, уровень можно наполнить объектами из внешнего файла с данными, т.е. получаем как раз то что нужно геймдизайнерам.

3. Шаблон

**Игровой мир** содержит **коллекцию объектов**. Каждый объект реализует **метод обновления, симулирующий один кадр поведения объекта.** На каждом кадреигра обновляет каждый объект из коллекции.

4. Когда использовать

Если шаблон Игровой цикл (Game Loop) можно сравнить с хлебом, то шаблон Метод обновления можно назвать маслом. В той или иной форме этот шаблон использует огромное число игр, в которых есть живые сущности. Если в игре есть космодесантники, драконы, марсиане, призраки или атлеты, скорее всего она использует этот шаблон.

Однако, если игра более абстрактная и движущиеся части в ней не намного живее, чем шахматные фигуры, этот шаблон может и не понадобиться. В играх наподобие шахмат вам нет необходимости моделировать поведение всех игровых объектов одновременно и, возможно, вам не придется просить пешку обновить себя на каждом кадре.

Вам не нужно обновлять их поведение на каждом кадре, но даже в настольной игре, вам возможно придется обновлять на каждом кадре анимацию. В этом вам шаблон тоже может помочь.

Метод обновления хорошо работает когда:

* В вашей игре есть некоторое количество объектов или систем, которые должны работать одновременно.
* Поведение каждого объекта практически не зависит от остальных.
* Объекты нужно обновлять постоянно.

**5. Имейте в виду**

Этот шаблон достаточно прост, чтобы скрывать в себе какие-то неприятные сюрпризы. Тем не мене каждая строка кода имеет последствия.

**5.1. Разделение кода на срезы отдельных кадров делает его сложнее**

Если вы сравните два первые примера кода, вы увидите, что второй уже значительно сложнее. Оба они заставляют скелет просто ходить туда-сюда, но второй делает это с перерывами на передачу управления коду игрового цикла на каждом кадре.

Такое изменение практически всегда необходимо для обработки пользовательского ввода, рендеринга и других вещей, о которых приходится заботиться игре, так что первый пример навряд ли можно назвать типичным. Зато он наглядно демонстрирует насколько усложняется поведенческий код, когда его приходится преобразовывать подобным образом.

**5.2. Вам нужно сохранять состояние, чтобы продолжать с того места, где вы прервались**

В первом примере кода у нас не было никаких переменных, определяющих налево идет стражник или направо. Это явно следовало из выполняющегося кода.

Когда мы поменяли этот код на код вида кадр за раз, нам пришлось добавить переменную patrollingLeft , чтобы за этим следить. Когда мы возвращаемся к нашему коду, предыдущая позиция теряется и нам нужно хранить достаточно информации чтобы восстановить состояние перед следующими кадром.

Здесь нам может помочь шаблон Состояние (State). Машины состояний (state machines) и их разновидности так часто встречаются в играх отчасти потому, что (что следует из их имени) они хранят состояние, которым вы можете воспользоваться после того, как отвлечетесь на что-то.

**5.3. Объекты симулируются на каждом кадре, но не в настоящем конкурентном режиме**

Объекты симулируются на каждом кадре, но не в настоящем конкурентном режиме В этом шаблоне игровой цикл перебирает всю коллекцию объектов и обновляет каждый из них. Внутри вызова update() , большинство объектов могут получить доступ ко всему остальному игровому миру, включая другие объекты, которые обновляются. Это значит, что порядок, в котором объекты обновляются, начинает иметь значение.

Если обновление A происходит перед B в списке обновления, тогда во время обновления A, оно видит предыдущее состояние B. Но когда обновляется B, оно уже видит A в новом состоянии, потому что A на этом кадре уже обновлялось. Даже если с точки зрения игрока все происходит одновременно, ядро игры все равно работает в пошаговом режиме. Просто один полный "шаг" соответствует по длительности одному кадру.

Если по какой либо причине, вы решили что ваша игра не должна работать в таком последовательном режиме, вам может помочь нечто наподобие шаблона Двойная буферизация (Double Buffer). В таком случае порядок обновления A и B перестанет играть какую-либо роль, потому что и тот и другой объект будут видеть предыдущее состояние другого.

Пока логика игры не предполагает тесного связывания это нормально. Обновление объектов в параллельном режиме таит в себе некоторые неприятные сюрпризы. Представьте себе шахматы, где белые и черные ходят одновременно. Обе стороны пытаются переставить фигуру в пустую клетку. Как же можно решить эту проблему?

Последовательное обновление эту проблему решает: каждое обновление постепенно изменяет мир из одного состояния в другое без промежуточного времени, когда вещи находятся в двойственном состоянии и требуют уточнения.

**5.4. Будьте осторожны при изменении списка объектов во время обновления**

Когда вы используете этот шаблон, на метод обновления обязательно завязывается слишком большая часть поведения игры. Обычно сюда относится и код, добавляющий и удаляющий обновляемые объекты в игре.

Например, наш скелет охранник будет дропать предмет после смерти. Новый объект вы можете без проблем просто добавить в конце списка обновляемых объектов. Вы идете по списку объектов дальше, доходите до только что добавленного нового объекта и обновляете и его тоже.

Только это совсем не значит, что новый объект имеет право действовать в том же кадре, когда его добавили, до того как игрок смог его хотя бы увидеть. Если вы не хотите чтобы это произошло, можно применить одну хитрость и кешировать в начале цикла обновления количество объектов в списке и обновлять только такое их количество:

int numObjectsThisTurn = numObjects\_;

for (int i = 0; i < numObjectsThisTurn; i++)

{

objects\_[i]->update();

}

Здесь objects\_ — это массив обновляемых объектов в игре, а numObjects\_ — его длина. Когда добавляется новый объект, длина увеличивается на единицу. Мы кэшируем длину в numObjectsThisTurn в начале цикла, так что итерация останавливается перед тем, как мы доберемся до объектов, добавленных на текущем кадре.

Проблема усложняется, если мы выполняем удаление во время итерации. Вы убиваете глупого монстра и теперь его нужно выкинуть из списка объектов. Если он находится в списке до объекта, который вы сейчас обновляете, вы можете случайно пропустить объект:

for (int i = 0; i < numObjects\_; i++)

{

objects\_[i]->update();

}

Этот простейший цикл инкрементирует индекс обновляемого объекта на каждой итерации. В левой части иллюстрации ниже показано как массив выглядит пока мы обновляем героиню.



Так как мы обновляем героиню, i равняется 1. Она убивает монстра и он удаляется из массива. Героиня перемещается на позицию 0 и несчастный крестьянин перемещается на позицию 1. После обновления героини, i инкрементируется до 2. Как вы можете видеть справа, несчастного крестьянина пропустили и он никогда не будет обновлен.

Простейшее решение — это двигаться по списку объектов в обратную сторону. Таким образом удаление объекта сместит только уже обновленные объекты.

Можно просто быть более аккуратным при удалении объектов и обновлять любые итерационные переменные, учитывая удаление. Или же можно подождать с удалением до тех пор, пока мы не обойдем весь список. Пометьте объект как "мертвый", но сразу не удаляйте. Во время обновления мы пропускаем мертвые объекты. А когда закончили — проходим список еще раз и удаляем все мертвые объекты.

Если вы обновляете объекты в цикле обновления в многопоточном режиме, вам тем более стоит повременить с любыми изменениями списка объектов, чтобы избежать расходов на синхронизацию во время обновления.

**6. Пример кода**

Шаблон настолько прямолинеен, что пример кода просто топчется на месте. Это совсем не значит, что шаблон бесполезен. Он очень **полезен**, потому что он **простой**: это просто решение проблемы без всяких украшений.

Но чтобы говорить более конкретно, давайте пройдемся по нескольким реализациям. Начнем с класса сущности, представляющей скелет или статую.

class Entity

{

public:

Entity()

: x\_(0), y\_(0)

{}

virtual ~Entity() {}

virtual void update() = 0;

double x() const { return x\_; }

double y() const { return y\_; }

void setX(double x) { x\_ = x; }

void setY(double y) { y\_ = y; }

private:

double x\_;

double y\_;

};

Я добавил в нее несколько вещей, но это самый минимум того, что понадобится нам в дальнейшем. Смею предположить что в реальном коде будет гораздо больше всяких связанных с графикой или физикой штук. Самое важное для нашего шаблона заключается в том, что у сущности есть абстрактный метод update().

Игра поддерживает коллекцию таких сущностей. В нашем примере мы поместим ее в класс, представляющий игровой мир:

class World

{

public:

World()

: numEntities\_(0)

{}

void gameLoop();

private:

Entity\* entities\_[MAX\_ENTITIES];

int numEntities\_;

};

Теперь, когда все готово, игра реализует шаблон, обновляя на каждом кадре все сущности:

void World::gameLoop()

{

while (true)

{

// Handle user input...

// Update each entity.

for (int i = 0; i < numEntities\_; i++)

{

entities\_[i]->update();

}

// Physics and rendering...

}

}

**7. Подклассы сущностей?!**

Уверен, что у некоторых читателей уже пошли мурашки по коже, когда они увидели, как я наследую от главного класса сущности для определения специального поведения. Если вы не видите в этом проблемы, я введу вас в курс дела.

Когда игровая индустрия вышла из первобытного супа ассемблерного кода 6502 и синхронизации с обратным ходом луча в мир объектно-ориентированных языков, разработчики попали в безумный мир архитектуры программного обеспечения. Одной из любимых возможностей было наследование. И была воздвигнута византийская башня наследования, такая высокая, что заслоняла собой солнце.

Это была ужасная идея и никто не мог поддерживать гигантскую гору иерархии классов без того, чтобы быть под ней погребенным. Даже банда четырех уже предупреждала об этом в еще 1994-м году:

*«Предпочитайте "композицию объектов" "наследованию классов"».*

Только между нами, мне кажется маятник слишком далеко качнулся от подклассов. Обычно я их избегаю, но не стоит возводить неиспользование наследования в догму, также как не стоит превращать в догму его использование. Используйте эту возможность осмотрительно и трезво оценивая последствия.

Когда это понимание проникло в игровую индустрию, решение виделось в шаблоне Компонент (Component). Если мы его применим, update() будет применяться к **компонентам сущности**, а не **к самой сущности**. Это позволяет избежать создания сложных иерархий классов сущностей для определения и повторного использования поведения. Вместо этого вы просто смешиваете и подбираете компоненты.

Если бы мы писали настоящую игру, я бы тоже так поступил. Но эта глава не про компонент. Она о методе update() и проще всего увидеть как он работает с минимальным количеством дополнительных деталей — это поместить метод непосредственно в Entity и унаследовать от нее несколько подклассов.

**8. Определение сущностей**

Хорошо, вернемся к нашей задаче. Изначально мы хотели реализовать патрулирование скелета-охранника и работу стреляющих молниями статуй. Начнем с нашего костлявого друга. Чтобы определить его патрульное поведение, создаем новую сущность с соответствующей реализацией update():

class Skeleton : public Entity

{

public:

Skeleton()

: patrollingLeft\_(false)

{}

virtual void update()

{

if (patrollingLeft\_)

{

setX(x() - 1);

if (x() == 0) patrollingLeft\_ = false;

}

else

{

setX(x() + 1);

if (x() == 100) patrollingLeft\_ = true;

}

}

private:

bool patrollingLeft\_;

};

Как вы видите, мы практически выдрали этот кусок кода из игрового цикла, который мы писали раньше и поместили его в метод update() класса Skeleton. Единственное серьезное отличие только в том, что patrollingLeft\_ превратилось в поле из обычной локальной переменной. Таким образом, значение будет сохраняться между вызовами update().

Давайте теперь перейдем к статуям:

class Statue : public Entity

{

public:

Statue(int delay)

: frames\_(0),

delay\_(delay)

{}

virtual void update()

{

if (++frames\_ == delay\_)

{

shootLightning();

// Reset the timer.

frames\_ = 0;

}

}

private:

int frames\_;

int delay\_;

void shootLightning()

{

// Shoot the lightning...

}

};

И опять мы практически просто перенесли сюда старый код из игрового цикла и кое- что переименовали. Таким образом, мы просто упрощаем кодовую базу. В оригинальном скученном императивном коде у нас были отдельные локальные переменные и для счетчика кадров статуи и для частоты стрельбы.

Теперь, когда мы перенесли их в сам класс Statue , вы можете создать столько переменных сколько нужно и у каждого экземпляра будет собственный таймер. Вот она истинная цель шаблона: **теперь нам гораздо проще добавлять новые сущности в игровой мир, потому что каждая из них располагает всем необходимым, чтобы о себе позаботиться**.

Шаблон позволяет нам **населять** мир, а не **реализовывать** его. А еще мы получаем дополнительную гибкость за счет наполнения мира через отдельный файл данных или редактор уровня.



**9. Ход времени**

Это была суть шаблона, но я хочу показать вам еще несколько вариаций. Пока что мы предполагали, что каждый вызов update() продвигает состояние игрового мира на фиксированный отрезок времени.

Мне такой подход нравится, однако многие игры используют переменный временной шаг. В этом случае игровой цикл симулирует более длинные или более короткие отрезки времени, в зависимости от того, сколько потребовалось времени на обработку и рендеринг предыдущего кадра.

Это значит что каждому вызову update() необходимо знать насколько продвинулись виртуальные часы. Поэтому внутрь передается количество прошедшего времени. Например, наш скелет патрульный может обрабатывать переменный временной шаг следующим образом:

void Skeleton::update(double elapsed)

{

if (patrollingLeft\_)

{

x -= elapsed;

if (x <= 0)

{

patrollingLeft\_ = false;

x = -x;

}

}

else

{

x += elapsed;

if (x >= 100)

{

patrollingLeft\_ = true;

x = 100 - (x - 100);

}

}

}

Теперь расстояние, проходимое скелетом увеличивается вместе с увеличением обрабатываемого времени. Как вы видите, работа с переменным временным отрезком увеличивает сложность кода. Скелет должен следить за тем, чтобы не выйти за рамки зоны патрулирования во время долгого временного отрезка и нам нужно корректно обрабатывать этот случай.

**10. Архитектурные решения**

**10.1. В каком классе будет жить метод обновления?**

Самое главное решение, которое вам нужно принять — это внутри какого класса разместить update().

* **Класс сущности:**

Это простейшее решение, если у вас уже есть класс сущности, потому что вам не нужно будет вводить в игру никаких дополнительных классов. Это может сработать, если у вас не слишком много видов сущностей, но в целом индустрия уходит от такого подхода.

Создавать подкласс Entity каждый раз когда вам нужно новое поведение — довольно хрупкий и болезненный метод, если видов сущностей у вас достаточно много. Вскоре вы обнаружите, что вам хочется повторно использовать куски кода, не слишком хорошо укладывающиеся в стройную иерархию и застрянете на этом.

* **Класс компонент:**

Если вы уже используете шаблон Компонент (Component), то тут и думать нечего. Он позволяет каждому компоненту обновляться независимо. Подобно тому как работает метод обновления для снижения связности в игре в целом, этот шаблон позволяет вам снизить связность частей отдельной сущности друг от друга. Рендеринг, физика и AI могут позаботиться о себе самостоятельно.

* **Класс делегат:**

Есть еще один шаблон, который предполагает делегирование части поведения класса другому объекту. Шаблон Cостояние (State) именно этим и занимается, так что вы можете менять поведение объекта, подменяя объект к которому он делегирует. Шаблон Объект тип (Type Object) делает это, позволяя вам разделять поведение между несколькими сущностями одного "вида".

Если вы используете один из этих шаблонов, поместить update() в класс, к которому мы делегируем, будет вполне логичным решением. В этом случае у вас все еще может быть метод update() в главном классе, но он уже будет не виртуальным, а просто будет вызывать объект делегат. Примерно так:

void Entity::update()

{

// Forward to state object.

state\_->update();

}

Это позволяет вам определять новое поведение, изменяя объект делегат. Как при использовании компонентов, он дает вам гибкость в изменении поведения без необходимости определять полностью новый подкласс.

**10.2. Как обрабатываются спящие объекты?**

Очень часто у вас в игровом мире присутствуют объекты, которые по какой-либо причине не нужно обновлять. Они могут быть отключенными, находящимися за пределами экрана или еще не разблокированными. Если у вас в этом состоянии находится довольно большое количество объектов, может быть довольно расточительно в плане процессорного времени проходить их все во время обновления каждого кадра.

Одним из решений является создание отдельной коллекции объектов только с "живыми" объектами, которые требуют обновления. Когда объект становится неактивным, он удаляется из этой коллекции. Когда включается снова — добавляется обратно. Таким образом мы будем проходить только по списку активных объектов.

**Если вы используете единственную коллекцию интерактивных объектов**:

**Вы впустую тратите время**. Для интерактивных объектов вам придется добавлять проверку флага в духе "активен ли я" или вызывать пустой метод.

В дополнение к потерянным процессорным тактам на проверку активности объекта, бесполезный перебор объектов приводит к ошибкам обращения к кешу. Процессор спроектирован таким образом, что быстрее читает данные из кеша, а не из памяти. Лучше всего это получается, когда следующий читаемый объект находится сразу за предыдущим.

Когда вы переходите к следующему объекту, вы можете пропустить оставшуюся в кеше порцию данных, заставляя процессор загружать следующую порцию данных из обычной памяти.

**Если вы используете отдельную коллекцию с активными объектами**:

**Вы используете дополнительную память для хранения второй коллекции**. В случае если у вас активны все объекты — это будет просто вторая копия первой коллекции. В таком случае она явно избыточна. Но когда скорость важнее памяти (а обычно так и есть) это допустимая потеря.

Еще одно решение — это перейти к двум коллекциям, но теперь во второй будут храниться только неактивные объекты.

**Вам нужно поддерживать синхронизацию коллекций**. Когда объект создается или полностью уничтожается (или временно отключается), вам нужно удалить или изменить его как в основной коллекции, так и в коллекции активных объектов.

Выбрать, что вам больше подходит, можно проанализировав, сколько неактивных объектов может у вас быть. Чем их больше, тем полезнее использовать для них отдельную коллекцию, чтобы не отделять их прямо внутри игрового цикла.