Шаблон «Двойная буферизация» (Double buffering)

1. Задача

2. Мотивация – зачем нам реализовывать задачу из первого пункта?

3. Как работает компьютерная графика (коротко) – как работает видео дисплей и компьютерный монитор в плане отрисовки картинки, что применяют в большинстве компьютеров, как это работает, какой баг отображения можно словить

4. Акт первый. Сцена первая – пример двойной буферизации с театром

5. Вернемся к графике – как здесь применяется подход из предыдущего пункта

6. Шаблон – что делает класс буфера, сколько хранит буферов, какие операции доступны, куда пишет, откуда считывает, как происходит обмен

7. Когда его использовать – 4 пункта

8. Подводные камни шаблона – 2 пункта

9. Пример кода – класс буфера и его поля и методы, класс сцены: с одним буфером и какая проблема может возникнуть и с двумя (код)

10. Искусственный интеллект – поведенческая система для игры по мотивам гротескной буффонады, класс актера (что за методы и поля у него есть и как они работают), класс сцены, на которой эти актеры находятся (тоже самое), подкласс комедианта (тоже самое), когда все нормально срабатывает, а когда ловим баг

11. Буферизация пощечин – как изменяем класс актера и метод в классе сцены

**1. Задача**

Дать возможность ряду последовательных операций выполнятся мгновенно или одновременно.

**2. Мотивация**

В своем сердце компьютер отсчитывает последовательность ударов. Его мощь заключается в способности разбивать громадные задания на мелкие шаги, которые можно выполнять один за другим. Однако пользователю зачастую нужно видеть, как вещи выполняются за один шаг или несколько задач выполняются одновременно.

*В потоковой и многоядерной архитектуре это уже не совсем верно, но даже при наличии нескольких ядер, всего только несколько операций могут выполняться в конкурентном режиме*.

Типичный пример, встречающийся в любом игровом движке — это рендеринг. Когда игра отрисовывает мир, видимый пользователем, она делает это отдельными кусочками: горы вдали, крутые холмы, деревья, все по очереди. Если пользователь увидит этот процесс отрисовки в таком инкрементном режиме, иллюзия когерентности мира теряется. Сцена должна обновляться плавно и быстро, образуя последовательность законченных кадров, появляющихся мгновенно.

Двойная буферизация решает эту проблему, но чтобы понять как, нам нужно для начала вспомнить как компьютер показывает графику.

**3. Как работает компьютерная графика (коротко)**

Видео дисплей, как и компьютерный монитор, рисует пиксель за пикселем. Они обновляются один за другим, слева направо в каждом ряду, а затем происходит переход вниз к следующему ряду. Когда нижний правый угол достигнут, происходит переход к левому верхнему углу и процесс начинается заново. Происходит это так быстро — по крайней мере, 60 раз в секунду, что наш глаз этого пробегания по рядам не замечает. Для нас все выглядит, так как будто сменяются статичные картинки на весь экран.

Можно думать об этом как о крошечном шланге, из которого мы поливаем экран пикселями. Отдельные цвета подводятся к этому шлангу и распыляются на экран по одному биту цвета за раз. Но каким образом этот шланг знает, куда какой цвет направлять?

В большинстве компьютеров ответ кроется в применение **буфера кадра** (framebuffer). **Буфер кадра** — это массив пикселей в памяти, участок RAM, где каждые несколько байтов представляют собой отдельный пиксель. И в то время когда шланг распыляет пиксели по экрану, он считывает значения цветов из массива по одному байту за раз.

Для того чтобы наша игра появилась на экране, все что мы на самом деле предпринимаем — это просто записываем значения в этот массив. Все самые изощренные графические алгоритмы сводятся к этому: установке значения байтов в буфере кадра. Но есть тут одна проблема.

Как я сказал ранее, компьютеры работают последовательно. Если машина выполняет кусок нашего кода рендеринга, мы не ожидаем что делаем в тот же самый момент что- то еще. В целом это верно, но некоторые вещи все-таки происходят во время выполнения нашей программы. Один из таких процессов — это постоянное считывание информации из буфера кадра. И это уже может быть проблемой.

Предположим что мы хотим отобразить на экране смайлик. Наша программа начинает в цикле двигаться по буферу кадра, окрашивая пиксели. Что мы до сих пор не уяснили — так это то, что видео драйвер производит считывание из буфера кадра в то же самое время, когда мы ведем в него запись. И когда он проходит по записанным пикселям, на экране начинает появляться смайлик. А потом он нас обгоняет и считывает данные из тех пикселей, куда мы еще ничего не записали. Результат паршивый: результатом будет баг, когда картинка отрисовывается только в верхней части экрана.



Мы начинаем отрисовывать пиксели из буфера кадра также как видео драйвер (Рис. 1). Видеодрайвер настигает рендер и попадает туда, куда пиксели еще не записывались (Рис. 2). Дале мы заканчиваем отрисовку (Рис. 3), но драйвер этого уже не видит. Вот результат, который пользователь увидит на экране (Рис. 4). Название "разрыв" возник потому что нижняя часть картинки как будто оторвана.

Вот здесь нам и пригодится наш шаблон. Наша программа рендерит пиксели по одному за раз, но драйверу нам нужно передавать всю картинку целиком — один кадр без смайлика и один кадр со смайликом. Именно для этого и нужна двойная буферизация.

**4. Акт первый. Сцена первая**

Представьте, что мы смотрим нашу собственную пьесу. Как только заканчивается первая сцена и начинается вторая, нам нужно сменить декорации. Если мы просто начнем перетаскивать реквизит, иллюзия когерентности пространства пропадет. Мы конечно можем просто приглушить свет на этот период (так в театре тоже делают), но аудитория по прежнему будет понимать что что-то происходит. Мы же хотим чтобы между сценами не было провалов.

В реальности мы можем прибегнуть к оригинальному решению: Построим две декорации таким образом, что они обе будут видны публике. У каждой сцены свое освещение. Назовем их A и B. Первая сцена демонстрируется в декорации A. В это время декорация B затемнена и работники сцены готовят ее к показу сцены два. Как только сцена первая завершается, мы выключаем свет в декорации A и включаем его в декорации B. Внимание публики сразу переключается к декорации B, где уже начинается сцена вторая.

В это время наши работники сцены занимаются затемненной декорацией A, подготавливая ее для сцены три. Как только сцена два закончится, мы снова переключим свет на декорацию A. Этот процесс будет повторяться на протяжении всей пьесы, используя затемненную декорацию для подготовки следующей сцены. Для перехода между сценами мы просто затемняем одну и освещаем другую. В результате наша публика получает возможность видеть спектакль без задержек между сценами. И никогда не видит работников сцены.

**5. Вернемся к графике**

Точно также работает и двойная буферизация, и именно такой процесс скрывается за системой рендеринга практически любой современной игры. Вместо единственного буфера кадра у нас есть два. Один из них представляет текущий кадр — аналогию декорации A. Это то место, откуда считывает данные видеодрайвер. GPU может производить из него считывание сколько угодно и когда угодно.

В это время наш код рендеринга пишет в другой буфер. Это наша затемненная декорация B. Когда код рендеринга заканчивает отрисовку сцены, мы переключаем свет, подменяя (swapping) буфера. Этим самым мы говорим видеобуферу, чтобы он теперь считывал данные из второго буфера вместо первого. И пока мы будем выполнять переключение в конце обновления экрана, никаких разрывов мы не увидим, и сцена будет отображаться целиком.

А в это самое время наш старый буфер кадра становится готовым к использованию. Мы начинаем рендерить в него новый кадр. Вуаля!

**6. Шаблон**

**Класс буфера** инкапсулирует **буфер** — часть состояния, которое можно изменить. Буфер изменяется постепенно, но мы хотим чтобы внешний код увидел изменение как единый атомарный процесс. Чтобы это стало возможным, класс хранит **два** буфера: **следующий** и **текущий**.

Когда требуется считать информацию из буфера — всегда используется текущий. А когда информация записывается — используется следующий буфер. Когда изменения закончены, операция **обмена** (swap) мгновенно меняет местами следующий и текущий буферы, так что новый буфер становится видным публично. Старый текущий буфер теперь доступен для повторного использования в качестве следующего буфера.

**7. Когда его использовать**

Это шаблон из тех, про который вы сами поймете, когда его нужно будет использовать. Если у вас есть система, в которой не хватает двойной буферизации, это обычно заметно (как в случае с разрывом) или приводит к некорректной работе. Но просто сказать "Вы поймете когда он вам пригодится" — недостаточно. Если говорить конкретнее, этот шаблон стоит применять если справедливо одно из следующих утверждений:

* У нас есть состояние, изменяющееся постепенно.
* К состоянию есть доступ посередине процесса его изменения.
* Мы хотим предотвратить код, считывающий состояние от чтения незаконченного изменения.
* Мы хотим иметь возможность считывать состояние, не дожидаясь, когда оно будет изменено.

**8. Подводные камни**

**Переключение само по себе требует времени**: двойная буферизация требует этапа переключения (swap), как только изменение будет закончено. Само это переключение должно быть атомарным — остальной код не должен иметь доступ во время этой операции ни к одному из состояний. Чаще всего переключение выполняется также быстро как переназначение указателя. А вот если переключение требует больше времени чем собственно изменение состояния, то толку от шаблона не будет никакого.

**Нам нужно иметь два буфера**: втрое следствие применения шаблона — увеличение потребления памяти. Как явственно следует из названия нам нужно постоянно держать именно две копии состояния в памяти. На устройствах с ограниченным объемом памяти это довольно дорогая цена за применение шаблона. Если вы не можете позволить себе иметь два буфера, вам стоит присмотреться к другим способам обеспечения недоступности состояния для чтения во время изменения.

**9. Пример кода**

Мы напишем очень приблизительную графическую систему вывода пикселей в буфер кадра. В большинстве консолей и PC всю эту низкоуровневую графическую работу делает видеодрайвер, однако ручная реализация позволит нам лучше разобраться в происходящем. Для начала сам буфер:

class Framebuffer

{

public:

Framebuffer() { clear(); }

void clear()

{

for (int i = 0; i < WIDTH \* HEIGHT; i++)

{

pixels\_[i] = WHITE;

}

}

void draw(int x, int y)

{

pixels\_[(WIDTH \* y) + x] = BLACK;

}

const char\* getPixels()

{

return pixels\_;

}

private:

static const int WIDTH = 160;

static const int HEIGHT = 120;

char pixels\_[WIDTH \* HEIGHT];

};

У него есть базовые операции очистки всего буфера в указанный цвет и установки цвета отдельного пикселя. Еще у него есть функция getPixels() , скрывающая за собой массив сырых данных в памяти, хранящий данные пикселей. Мы не увидим ее в примере, но видеодрайвер будет часто использовать такую функцию для пересылки содержимого буфера на экран.

Обернем этот сырой буфер классом Scene . Его задача заключается в отрисовке чего- либо с помощью вызовов draw() своего буфера:

class Scene

{

public:

void draw()

{

buffer\_.clear();

buffer\_.draw(1, 1);

buffer\_.draw(4, 1);

buffer\_.draw(1, 3);

buffer\_.draw(2, 4);

buffer\_.draw(3, 4);

buffer\_.draw(4, 3);

}

Framebuffer& getBuffer() { return buffer\_; }

private:

Framebuffer buffer\_;

};

На каждом кадре игра командует сцене отрисоваться. Сцена очищает буфер и затем отрисовывает один за другим кучу пикселей. Еще она предоставляет доступ к внутреннему буферу через getBuffer() , чтобы видеодрайвер мог получить нужные ему данные.

Выглядит достаточно прямолинейно, но если оставить все как есть, у нас будут серьезные проблемы. Проблема в том, что видеодрайвер может вызвать getPixels() у буфера в любое время, даже здесь:

buffer\_.draw(1, 1);

buffer\_.draw(4, 1);

// <- Video driver reads pixels here!

buffer\_.draw(1, 3);

buffer\_.draw(2, 4);

buffer\_.draw(3, 4);

buffer\_.draw(4, 3);

Когда такое происходит, пользователь увидит глаза на лице, а рот на один кадр пропадет. На следующем кадре отрисовка прервется в какой-либо еще точке. В результате у нас получится ужасно моргающая графика. Исправить это можно добавлением второго буфера.

class Scene

{

public:

Scene()

: current\_(&buffers\_[0]),

next\_(&buffers\_[1])

{}

void draw()

{

next\_->clear();

next\_->draw(1, 1);

// ...

next\_->draw(4, 3);

swap();

}

Framebuffer& getBuffer() { return \*current\_; }

private:

void swap()

{

// Just switch the pointers.

Framebuffer\* temp = current\_;

current\_ = next\_;

next\_ = temp;

}

Framebuffer buffers\_[2];

Framebuffer\* current\_;

Framebuffer\* next\_;

};

Теперь у Scene есть два буфера, хранящиеся в массиве buffers\_ . Мы не ссылаемся на них из массива напрямую. Вместо этого у нас есть два члена класса next\_ и current\_ , указывающие на массив. Когда мы рисуем, мы выполняем отрисовку на следующий буфер, на который ссылается next\_ . А когда видеодрайверу нужно считать значение пикселя, он всегда обращается к другому буферу через current\_ .

Таким образом, видеодрайвер никогда не видит буфер, с которым мы в данный момент работаем. Единственный оставшийся кусочек пазла — это вызов swap() после того, как сцена заканчивает отрисовывать кадр. Он меняет местами два буфера, просто обменивая между собой указатели в next\_ и current\_. В следующий раз, когда видеодрайвер вызовет getBuffer(), он обратится к новому буферу, в который мы только что закончили рисовать и выведет последний кадр на экран. И никаких больше разрывов и неприятных глитчей.

**10. Искусственный интеллект**

Давайте представим себе, что мы разрабатываем поведенческую систему для игры по мотивам гротескной буффонады . В игре имеется сцена, в которой участвует куча актеров, творящих всякие шутки и трюки. Вот базовый актер:

class Actor

{

public:

Actor() : slapped\_(false) {}

virtual ~Actor() {}

virtual void update() = 0;

void reset() { slapped\_ = false; }

void slap() { slapped\_ = true; }

bool wasSlapped() { return slapped\_; }

private:

bool slapped\_;

};

На каждом кадре игра отвечает за то чтобы вызвать update() актера. Таким образом он может что-либо сделать. С точки зрения игрока критически важно, чтобы обновления всех актеров выглядели одновременными.

Кроме этого, актеры могут взаимодействовать друг с другом и под "взаимодействовать" я понимаю "давать друг другу пощечины". Во время обновления актер вызывает метод slap() другого актера, чтобы дать ему пощечину и вызывает wasSlapped() , чтобы определить получил ли пощечину сам.

Актерам потребуется декорация, в которой они будут взаимодействовать:

class Stage

{

public:

void add(Actor\* actor, int index)

{

actors\_[index] = actor;

}

void update()

{

for (int i = 0; i < NUM\_ACTORS; i++)

{

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

actors\_[i]->reset();

}

}

private:

static const int NUM\_ACTORS = 3;

Actor\* actors\_[NUM\_ACTORS];

};

Scene позволяет нам добавлять актеров и предоставляет единый вызов update() , обновляющий всех актеров. Несмотря на то, что для зрителя актеры выглядят действующими одновременно, на самом деле они обновляются один за другим.

Единственное, о чем нужно упомянуть — это то, что каждое состояние "получил пощечину" очищается сразу после обновления. Это сделано для того, чтобы каждый актер реагировал на пощечину только один раз.

Чтобы все заработало, давайте определим конкретный подкласс актера. Наш комедиант довольно прост. Он находится напротив другого актера. Когда он получает пощечину (от кого угодно) — он реагирует на пощечину актера, который находится перед ним.

class Comedian : public Actor

{

public:

void face(Actor\* actor) { facing\_ = actor; }

virtual void update()

{

if (wasSlapped()) facing\_->slap();

}

private:

Actor\* facing\_;

};

Теперь запустим в декорацию несколько комедиантов и посмотрим, что получится. Добавим трех комедиантов, каждый из которых смотрит на следующего. Последний смотрит на первого, замыкая получившийся круг:

Stage stage;

Comedian\* harry = new Comedian();

Comedian\* baldy = new Comedian();

Comedian\* chump = new Comedian();

harry->face(baldy);

baldy->face(chump);

chump->face(harry);

stage.add(harry, 0);

stage.add(baldy, 1);

stage.add(chump, 2);

Получившаяся декорация выглядит следующим образом. Стрелки показывают кто на кого смотрит, а номера обозначают индекс в массиве декорации.



Дадим пощечину Гарри и посмотрим что из этого получится, когда мы запустим обновление.

harry->slap();

stage.update();

Помните, что функция декорации update() обновляет актеров по очереди, так, что если мы проследим, что происходит в коде, мы обнаружим следующее:

Stage updates actor 0 (Harry)

Harry was slapped, so he slaps Baldy

Stage updates actor 1 (Baldy)

Baldy was slapped, so he slaps Chump

Stage updates actor 2 (Chump)

Chump was slapped, so he slaps Harry

Stage update ends

Итак, в единственном кадре наша начальная пощечина Гарри прошлась по всем комедиантам. Теперь, чтобы немного разнообразить ситуацию, мы поменяем актеров в массиве декорации местами, но смотреть они будут друг на друга по-прежнему.



Не будем трогать остальную часть декорации, а просто заменим код с добавлением актеров на следующий:

stage.add(harry, 2);

stage.add(baldy, 1);

stage.add(chump, 0);

Давайте посмотрим что произойдет, когда мы запустим наш эксперимент снова:

Stage updates actor 0 (Chump)

Chump was not slapped, so he does nothing

Stage updates actor 1 (Baldy)

Baldy was not slapped, so he does nothing

Stage updates actor 2 (Harry)

Harry was slapped, so he slaps Baldy

Stage update ends

Ух ты! Совсем другое дело. Проблема очевидна. Когда мы обновляем актеров, мы изменяем состояние "получил пощечину", т.е. то же самое состояние, которое мы читаем во время обновления. Из—за этого сделанные вначале процесса обновления изменения начинают влиять на то, что происходит дальше в процессе того же самого шага обновления.

В конце концов, получается так, что актеры начинают реагировать на пощечину либо на том же самом кадре, либо на следующем только на основании того, в какой очередности они находятся в декорации. Это нарушает наше главное требование — обеспечение видимости одновременного действия всех актеров: порядок, в котором расположены актеры, не должен влиять на результаты обновления в каждом кадре.

**11. Буферизация пощечин**

К счастью, нам может помочь наш шаблон Двойной буферизации. На этот раз, вместо того, чтобы заводить две монолитные копии "буферизуемого" объекта, мы буферизуем гораздо более мелкую сущность — состояние "получил пощечину" у каждого из актеров:

class Actor

{

public:

Actor() : currentSlapped\_(false) {}

virtual ~Actor() {}

virtual void update() = 0;

void swap()

{

// Swap the buffer.

currentSlapped\_ = nextSlapped\_;

// Clear the new "next" buffer.

nextSlapped\_ = false;

}

void slap() { nextSlapped\_ = true; }

bool wasSlapped() { return currentSlapped\_; }

private:

bool currentSlapped\_;

bool nextSlapped\_;

};

Вместо единственного состояния slapped\_ , у каждого актера будет два. Как и в предыдущем графическом примере, текущее состояние используется для считывания, а следующее — для записи.

**Функция reset() заменяется функцией swap()**. Теперь прямо перед очисткой подменяемого состояния, оно копирует следующее состояние в текущее и делает его новым текущим. Для этого требуется внести небольшое изменение в Stage:

void Stage::update()

{

for (int i = 0; i < NUM\_ACTORS; i++)

{

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

}

for (int i = 0; i < NUM\_ACTORS; i++)

{

actors\_[i]->swap();

}

}

Функция update() теперь обновляет всех актеров и только после этого подменяет все их состояния.

В конце концов, у нас получится такая система, в которой актер видит пощечину только на следующем кадре после того, как она была нанесена. И в этом случае актеры будут действовать одинаково, вне зависимости от порядка нахождения в массиве. Для пользователя или внешнего кода обновление актеров будет выглядеть одновременным.