Игровой цикл (Game Loop)

1. Задача

2. Циклы событий – как выглядит цикл для текстового процесса, что он ожидает, что происходит с программой в этот момент, чем отличаются игры, что происходит в играх с пользовательским вводом и что в играх происходит на каждом этапе игрового цикла

3. Мир вне времени – если цикл не блокируется, то какой возникает вопрос? Что происходит с точки зрения двух участников игры (по обе стороны экрана)? Что получаем за характеристику скорости работы игры? Что определяет в нашем игровом цикле эту характеристику (два фактора)?

4. Секунды в секунду – какой фактор фиксирован? Что происходит со старыми играми при запуске на различных устройствах? Как сейчас с этим обстоят дела у разработчиков игр? Еще одна важная особенность игрового цикла

5. Шаблон – когда работает и за что отвечает (4 пункта)

6. Пример кода (вступление небольшое) – рассматриваем ли мы в данном случае функции, вызываемые в игровом цикле

7. Беги, беги настолько быстро, насколько сможешь – в чем проблема простого подхода?

8. Передохнем немного – каким способом решаем исправить ситуацию (связан с FPS), однако почему он плохо работает?

9. Один маленький шаг и один гигантский шаг – на какие две проблемы разбивается та, с которой мы столкнулись? Что будет наблюдаться при преобладании одного над другим в разных случаях? С помощью какой идеи (и кода) мы решаем проблему, как эта идея работает? Что мы получаем в итоге? Однако, какая возникает новая проблема?

10. Догонялки – рендеринг, что на него не влияет, что мы можем с ним сделать? Как изменяем нашу логику и код? На величину какой переменной нужно обращать внимание и почему? В чем главная хитрость, которую мы применили?

11. Застрять в середине – последняя проблема, в чем заключается (как мы обновляем игру и осуществляем рендеринг)? Пример с пулей, как исправить ситуацию (спасение уже есть в нашем коде)?

**1. Задача**

Устранить зависимость игрового времени от пользовательского ввода и скорости процессора

**2. Циклы событий**

Современные программы с графическим пользовательским интерфейсом, если снять с них оболочку, поразительно напоминают старые текстовые квесты. Ваш текстовый процессор обычно просто сидит и ничего не делает до тех пор, пока вы не нажмете какую-либо клавишу:

while (true)

{

Event\* event = waitForEvent();

dispatchEvent(event);

}

Единственное различие здесь в том, что вместо текстовых команд, программа ожидает пользовательского ввода — нажатия мыши и клавиш. Но в основе лежит тоже самое, что и в текстовых квестах, где программа блокируется в ожидании пользовательского ввода, что на самом деле является проблемой.

В отличие от других программ, игры продолжают работать даже когда пользователь не предоставляет никакого ввода. Если вы будете просто смотреть на экран, игра не остановится. Анимации продолжат проигрываться. Визуальные эффекты танцуют и блестят. А если вам не повезет, монстр продолжит понемногу грызть вашего героя. Вот мы и подошли к первой ключевой особенности игрового цикла: он обрабатывает пользовательский ввод, но не ожидает его. Цикл продолжает крутиться всегда:

while (true)

{

processInput();

update();

render();

}

Позже мы рассмотрим его подробнее, но в основе лежат именно эти вещи. processInput() обрабатывает пользовательский ввод с момента прошлого вызова. update() продвигает симуляцию игры на один шаг. Сюда входят ИИ и физика (обычно именно в таком порядке). И наконец render() рисует игру, чтобы игрок увидел происходящее.

**3. Мир вне времени**

Если этот цикл не блокируется при вводе — возникает резонный вопрос: насколько **быстро** он крутится? На каждом шаге игрового цикла состояния игры немного продвигается вперед. С точки зрения обитателей игрового мира это заставляет их время идти вперед.

Обычно единица игрового цикла называется «тиком» или «кадром».

При этом у **игрока** время тоже тикает. Если мы измерим скорость выполнения циклов в единицах реального времени, мы получим единицу измерения «кадры в секунду» или FPS . Если игровые циклы сменяются быстро, FPS высок, а игра работает быстро и плавно. Если медленно — игра подтормаживает и становится похожей на кино в замедленном воспроизведении.

В нашем примитивном игровом цикле, который старается сменяться как можно чаще, частоту кадров определяют два фактора. Первый — это **сколько работы нужно выполнять на каждом кадре**. Сложная физика, куча игровых объектов и детализированная графика могут настолько нагрузить ваши процессор и видеокарту, что на обработку кадра понадобится очень много времени.

Второй фактор — это **производительность платформы**. Быстрые чипы могут перемалывать гораздо больше кода за то же время. Количество ядер, видеокарта, дискретный аудио чип и планировщик ОС — все это влияет на количество действий, которые можно успеть выполнить за один тик.

**4. Секунды в секунду**

В каждой игре второй фактор всегда фиксирован. Если вы пишете игру для NES или Apple II , вы точно знаете какой процессор у вас будет и можете на него рассчитывать. Все о чем вам нужно заботиться — это каким объемом работы вы его нагружаете.

Старые игры разрабатывались таким образом, чтобы выполняемая работа позволяла игре работать с нужной скоростью. Но если бы вы попробовали поиграть в ту же самую игру на более быстром или медленном компьютере, сама игра стала бы работать быстрее или медленнее.

Сейчас редко кто из разработчиков может позволить себе роскошь знать на каком железе будет работать их игра. Вместо этого играм приходится адаптироваться под большое разнообразие конфигураций.

Таким образом мы подошли еще к одной важной особенности игрового цикла: он обеспечивает постоянную скорость игры в независимости от аппаратного обеспечения.

**5. Шаблон**

**Игровой цикл** работает на протяжении всей игры. На каждом своем цикле игровой цикл **обрабатывает пользовательский ввод** без блокировки, **обновляет состояние игры** и **рендерит игру**. А еще он следит за ходом времени и **управляет скоростью игрового процесса**.

**6. Пример кода**

Игровой цикл управляет ИИ , рендерингом и другими игровыми системами, но они ни в коей мере не являются частью самого шаблона. Они просто из него вызываются. Так что реализация (непростая!) render() , update() и остальных оставляется читателю.

**7. Беги, беги настолько быстро, насколько можешь**

Мы уже видели самый примитивный игровой цикл:

while (true)

{

processInput();

update();

render();

}

Его главная проблема в том, что он никак не управляет скоростью игры. На быстрой машине он будет работать так быстро, что пользователь даже не разберет, что происходит. На медленной машине игра будет просто тормозить. Если в какой-то части игры у вас есть сложный ИИ или физика, игра тоже будет замедляться в этих местах.

**8. Передохнем немного**

Первый вариант, который мы рассмотрим, добавляет простой фикс. Скажем мы хотим чтобы наша игра работала с FPS 60. Это около 16 миллисекунд на кадр.

1000 мс / 60 FPS = 16 мс на кадр

Пока вы сможете уложить все свои вычисления и рендеринг в это время, у вас будет постоянный фреймрейт. Все что вам нужно будет сделать — это после обработки каждого кадра ожидать, пока наступит время для следующего примерно таким образом:



Код изменится таким образом:

while (true)

{

double start = getCurrentTime();

processInput();

update();

render();

sleep(start + MS\_PER\_FRAME - getCurrentTime());

}

sleep() здесь контролирует, чтобы игра не работала слишком быстро, если успела обработать кадр раньше времени. Но это не поможет, если ваша игра работает слишком медленно. Если игра будет тратить на цикл больше 16 мс, время сна станет отрицательным. Если бы у нас был компьютер, способный перемешаться во времени в прошлое, все было бы просто, но увы.

Вместо этого игра просто замедляется. Вы можете избежать этого делая меньше работы на каждом цикле: урезать и упростить графику или оглупить ИИ . Но это повлияет на качество игрового процесса даже на быстрых машинах.

**9. Один маленький шаг и один гигантский шаг**

Давайте попробуем кое-что посложнее. Проблема, с которой мы столкнулись, разбивается на следующие:

1. На каждом цикле время игры немного продвигается вперед.

2. Этот процесс требует некоторого количества реального времени.

Если шаг второй займет больше времени, чем первый, игра замедлится. Если на то, чтобы продвинуть время в игре на 16 мс придется потратить больше 16 мс — это не годится. А вот если за один шаг мы сможем обработать больше чем 16 мс игры, мы сможем обновлять игру уже реже и при этом не отставать.

Идея выбора длины шага основывается на количестве реального времени, прошедшего с прошлого кадра. Чем больше времени занял кадр, тем б**о**льшиешаги делает игра. Мы всегда будем успевать за реальным временем потому что будем выполнять все б**о**льшие и б**о**льшие шаг. Такое время шага можно назвать переменным или гибким. Выглядит это следующим образом:

double lastTime = getCurrentTime();

while (true)

{

double current = getCurrentTime();

double elapsed = current - lastTime;

processInput();

update(elapsed);

render();

lastTime = current;

}

На каждом кадре мы определяем сколько реального времени прошло с момента последнего обновления игры ( elapsed ). А когда мы обновляем состояние игры, мы передаем это значение внутрь. Далее движок отвечает за то, чтобы обновить игру на переданное количество времени.

Представим, что у нас через весь экран летит пуля. При фиксированной длительности шага, на каждом кадре она будет двигаться в соответствии со своей скоростью. С переменой длительностью шага вам придется масштабировать скорость в зависимости от прошедшего времени. Как только шаг увеличивается, пуля летит быстрее, чем на прошлом кадре. Главное, что экран целиком пуля пролетит за одно и то же количество реального времени, независимо от того, произойдет это за 20 маленьких быстрых шагов или за четыре больших медленных. Похоже на то, что нужно:

Игра работает с одинаковой скоростью на любом оборудовании.

Игроки с более мощными компьютерами вознаграждаются более плавным геймплеем.

Но, к сожалению, у нас появляется и одна серьезная проблема: игра получается недетерминированной и нестабильной. Вот пример того, в какую ловушку мы угодили:

Предположим, у нас идет сетевая игра, и Фред играет на ультрамощном PC, а Джордж на старом нетбуке своей бабушки. Упомянутая выше пуля пролетает по экрану у них обеих. На компьютере Фреда игра летает и каждый шаг занимает очень мало времени. У него пуля пролетает экран за секунду и происходит это за 50 кадров. У бедняги Джорджа на весь этот процесс отводится только пять кадров.

Это значит, что на компьютере Фреда физический движок обновляет позицию пули 50 раз в секунду, а у Джорджа — всего пять. Большинство игр используют при расчетах вещественные числа, которые очень подвержены ошибкам округления. Каждый раз, когда вы складываете два вещественных числа, результат получается немного другим.

Машина Фреда работает в десять раз быстрее и на ней накапливается большая ошибка чем у Джорджа. В результате через секунду полета пуля окажется на этих двух машинах в разных местах.

Это одна из самых неприятных проблем, возникающих при переменном временном шаге, но есть и другие. Для того, чтобы физический движок работал в реальном времени, правила реальной механики приходится аппроксимировать. Чтобы эти аппроксимации «не взорвались», их приходится глушить. Такое приглушение требует крайне чувствительной подстройки к конкретному временному шагу. Стоит изменить этот шаг, и физика станет нестабильной.

**10. Догонялки**

Обычно рендеринг — это та часть движка, на которую переменное время не влияет. Так как рендеринг происходит в конкретное время, для него не важно сколько времени прошло с момента прошлого рендеринга. Вещи рендерятся в том состоянии, в котором находятся в данный момент.

Мы можем использовать этот факт в свою пользу. **Обновлять** игру мы будем фиксированными шагами, потому что это проще и в плане ИИ и физики гораздо стабильнее. Но вот в плане **рендеринга** мы позволим себе некоторую гибкость для того, чтобы сэкономить немного процессорного времени.

Поступим следующим образом. С момента окончания прошлого игрового цикла у нас прошло некоторое время. Именно столько времени нам нужно просимулировать в игре, чтобы отобразить текущее состояние игроку. Добьемся мы это с помощью серии **фиксированных** временных шагов. У нас получится такой код:

double previous = getCurrentTime();

double lag = 0.0;

while (true)

{

double current = getCurrentTime();

double elapsed = current - previous;

previous = current;

lag += elapsed;

processInput();

while (lag >= MS\_PER\_UPDATE)

{

update();

lag -= MS\_PER\_UPDATE;

}

render();

}

Он состоит из нескольких частей. В начале каждого кадра мы обновляем lag на основе прошедшего реального времени. Это значение обозначает насколько наше игровое время отстало от реального. Далее мы будем обновлять состояние игры шагами фиксированной длины до тех пор, пока не догоним реальное время. Как только мы его догнали, выполняем рендеринг и начинаем процедуру снова. Визуализировать это можно следующим образом:



Обратите внимание, что временной шаг теперь не соответствует видимому фреймрейту. MS\_PER\_UPDATE — это **дробность** наших обновлений игры. Чем короче этот шаг, тем больше процессорного времени нужно для того, чтобы нагнать реальное время. Чем он длиннее — тем грубее геймплей. В идеале он должен быть довольно коротким, чтобы FPS был не ниже 60 и игра на быстрых машинах работала плавно.

Но будьте осторожны и не сделайте его слишком коротким. Нам нужно быть уверенными, что временной шаг будет больше, чем время, необходимое на обработку update() даже на медленном оборудовании. В противном случае игра всегда будет опаздывать и никогда не догонит реальное время.

К счастью, теперь у нас есть пища для размышлений. Хитрость здесь в том, что мы **выдернули рендеринг из цикла обновлений**. Это освобождает кучу процессорного времени. В результате сама игра симулируется с константной скоростью обновления на самом разном оборудовании. А если игрок видит подтормаживания на слабой машине, то подтормаживает только видимая часть игры.

Я на этом остановлюсь, но вы можете подстраховаться, ограничив количество временных шагов, которые могут выполняться друг за другом некоторым максимумом. Игра замедлится, но по крайней мере не заблокируется насовсем.

**11. Застрять в середине**

У нас осталась одна проблема и это остаточный лаг. Мы обновляем игру фиксированными временными шагами, а рендеринг выполняем в произвольные моменты времени. Это значит, что с точки зрения игрока, игра будет часто рендериться между двумя обновлениями.

Посмотрите на таймлайн:



Как вы видите, мы выполняем обновления довольно частыми, фиксированными интервалами. И время от времени выполняем рендеринг. Он выполняется реже, чем обновления и с не очень стабильным интервалом. И то и другое приемлемо. Не очень хорошо то, что мы не всегда выполняем рендеринг сразу после обновления. Посмотрите на время третьего рендеринга. Он оказался прямо между двумя обновлениями:



Представьте себе летящую через экран пулю. На момент первого обновления она находится слева. На момент второго — справа. Рендеринг произошел ровно посередине между этими двумя событиями, так что игрок может ожидать увидеть пулю прямо в центре экрана. Но в нашей реализации она по-прежнему будет находиться слева. И поэтому движение будет смотреться дерганным и запинающимся.

На наше счастье, мы точно знаем в какой момент между обновлениями происходит рендеринг: значение хранится в lag. Мы пережили предыдущее обновление, с тех пор прошло меньше времени, чем наш шаг обновления и длительность этого времени больше нуля. Так где мы оказались? Мы слегка залезли в следующее обновление.

Когда мы попадаем в рендер, мы передаем следующее:

render(lag / MS\_PER\_UPDATE);

Наш рендер обладает информацией обо всех объектах и их текущей скорости. Скажем пуля находится на 20 пикселей от левого края экрана и двигается со скоростью 400 пикселей за кадр. Если мы на полпути до следующего обновления, мы должны передать в render() 0.5. И пулю мы соответственно рисуем в позиции 220 пикселей. Та-дам! Плавное движение.

Мы разделили здесь MS\_PER\_UPDATE чтобы получить нормализованное значение. Значение, передаваемое в render() может варьироваться от 0(сразу после прошлого обновления) до 1(прямо перед следующим обновлением), независимо от временного шага обновления. Поэтому рендеру не нужно волноваться о фреймрейте. Он просто оперирует значениями от 0 до 1.

Конечно такая экстраполяция тоже может быть ошибочной. Когда мы на самом деле будем просчитывать следующий кадр, мы можем обнаружить что пуля наткнулась на препятствие или замедлилась чем-то. А мы уже рендерили ее в позиции, полученной интерполяцией предыдущего положения и предполагаемого будущего. Но верно это или нет мы сможем узнать только на следующем полном обновлении физики и ИИ.

Не удивительно, что такие попытки игры в угадывание не всегда заканчиваются успешно. К счастью, они обычно малозаметны. И по крайней мере, они не так заметны, как запинания, которые можно увидеть, если вообще не применять интерполяцию.