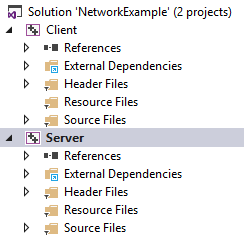
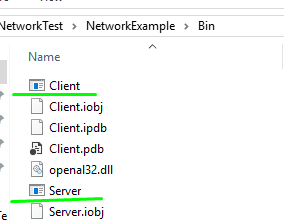
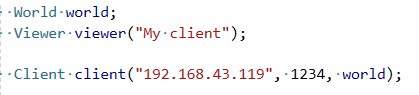
1. В рамках одного Solution можно делать несколько проектов:



Каждый проект генерирует свой артефект:



1. Посмотрим на код клиента (ф-я int main()):



**Игровой мир** – описание всех объектов и игровой логике в игре. Источник правды – игровой мир на сервере, т.к. он знает самую актуальную информацию о всех игроках. Сервер принимает решение, если один игрок «подстелил» другого, мяч попал в ворота и т.п. Игровой мир на стороне клиентов нужен: 1) для хранения состояния объектов и понимания, где и что отрисовывать 2) если объекты движутся предсказуемо (например, игроки не нажимали никакие другие кнопки клавиатуры/мыши/джойстика, и никто ни в кого не врезался, гол не забил …) то нет смысла нагружать сеть и лишний раз серверу слать что-то клиентам … клиенты сами в состоянии предсказать состояние игры в следующий момент времени.

**Viewer** – класс для отображения игрового мира

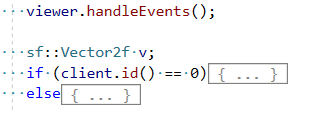
**Client** – класс для синхронизации: отправляет на сервер все действия игрока (был произведен выстрел … сменилось направление движения или скорость и т.п.) + принимает от сервера актуальное состояние игрового мира – т.к. одновременно другие игроки могут воздействовать на него.

Логика клиента такая:

пока окно открыто и client в состоянии общаться с сервером



обрабатываем все события произошедшие (кто-то куда-то кликнул мышью, изменил размер окна, понажимал кнопки):



Для удобства разработки поддержана возможность запуска нескольких клиентов на одной машине … и управления разными игроками с одной клавиатуры.

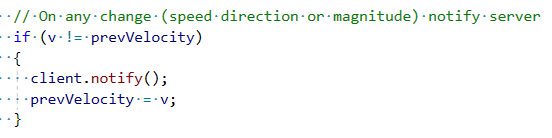
В примере единственное, что может делать игрок – менять **свое** направление движения



Дальше мы просчитываем, что будет с игровым миром (кто и где будет находиться) в следующий момент времени t+dt:



Если что-то изменилось, то информируем сервер об этом



Перерисовываем игровой мир на каждом кадре, чтобы у игрока на мониторе всё плавно двигалось:

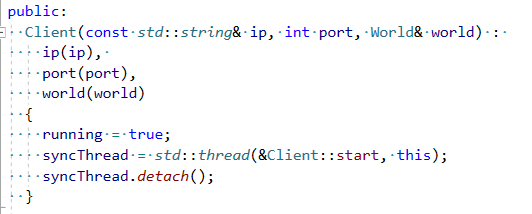


Классы **World** и **Viewer** не сложные. Не будем останавливаться на них.

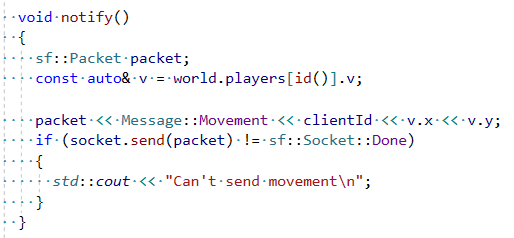
**Client** должен одновременно иметь возможность отправить на сервер свои изменения и получить с сервера актуальные данные про всех остальных клиентов.

Чтобы прием и отправка работали параллельно, запускаем каждую операцию в своей нити … ровно точно также делала «чат» в прошлом семестре.

Т.е. в отдельной нити создаем соединение с сервером и начинаем получать сообщения от него:



Из основной же нити отправляем оповещения о каких-то изменениях на стороне клиента:



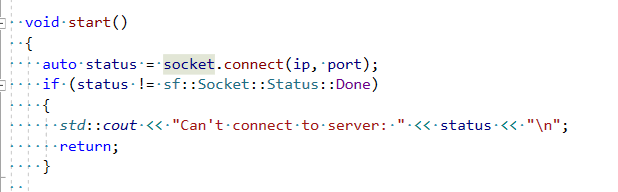
Как видно оправляем «пакет» с данными в формате: тип сообщения, идентификатор клиента, данные … в зависимости от типа сообщения принимающая сторона поймет, как пришедшие данные надо интерпретировать.

Сокет здесь ничем не отличается идейно от сокетов из 3го семестра:



В примере использовал tcp сокет – хорошо работает, не тормозит. В проектах можете и upd использовать.

Старт клиента начинается с подключения в серверу:



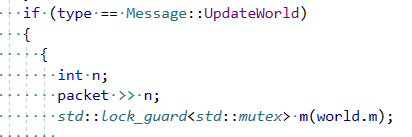
Клиенты обязаны знать ip и port, на которых работает сервер.



Дальше в цикле по мере прихода сообщений от сервера, происходит их обработка.

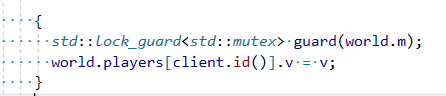
Могут прийти два типа сообщений: СlientCreated – сервер поставил нового только подключившегося клиента на случайное положение на карте и присвоил клиенту идентефикатор … и оповестил его о этом.

UpdateWorld – это означает нужно провести обновить состояние мира на стороне клиента более актуальными данными.



Помните что такое «состояние гонки» из прошлого семестра? Это когда параллельно из разных потоков происходит изменение каких-то данных … что может привести их в не консистентное состояние. И мы боролись с этим в 3м семестре с использованием семафоров.

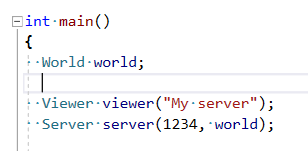
Здесь std::lock\_guard по сути делаем гарантирует, что



хоть мы и модифицируем игровой мир из разных потоков, но состояния гонки мы не получим.

С клиентом разобрались. Переходим к серверу.

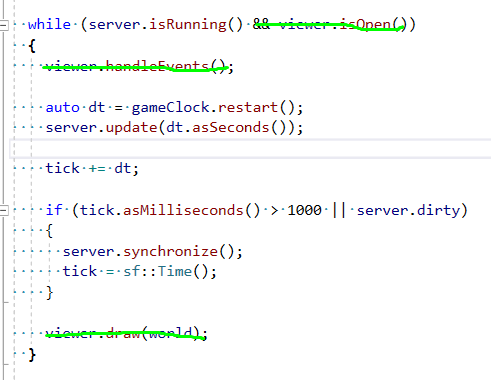
Ф-я int main сервера очень похожа на то, что было у клиента:



Игровой мир – хранение актуального состояния игры. **Server** – обеспечивает синхронизацию между клиентами.

**Viewer** фактически нужен здесь только для отладки, чтобы убеждаться, что синхронизация происходит корректно, сообщения от клиентов доходят. Очень рекомендую для целей отладки визуализировать также всё и на стороне сервера. Класс Viewer используется ровно тот же, даже лишний код писать не нужно. Но даже, если нужно, то визуализации отладочной информации сэкономит вам очень много времени.

Как мы поняли Viewer на сервере вещь не обязательная, так что остается по сути строк 10 кода:



Пока сервер работает, 1) обновляем игровой мир на dt 2) если очень давно не отправляли сообщение синхронизации клиентам



Либо произошло какое-то важное событие (один из игроков стрельнул в другого и второй должен знать, что в него пуля летит и т.п.)



То рассылаем на всех обновленное состояние игрового мира и сбрасываем счётчик «последней синхронизации»



Класс **World** общий с клиентом. Нужно по сути разобраться, как работает класс **Server**.

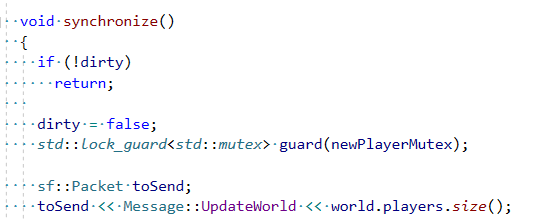
Серверу нужно параллельно уметь отправлять сообщения клиентам и принимать от них нотификации –> работаем в 2 потока.

В основном потоке рассылаем сообщения синхронизации клиентам. В дополнительном получаем события от клиентов о изменении направления движения и т.п.



Начинаем случать заданный порт и стартуем нить для получения сообщений от клиентов.

Из основного потока вызывается

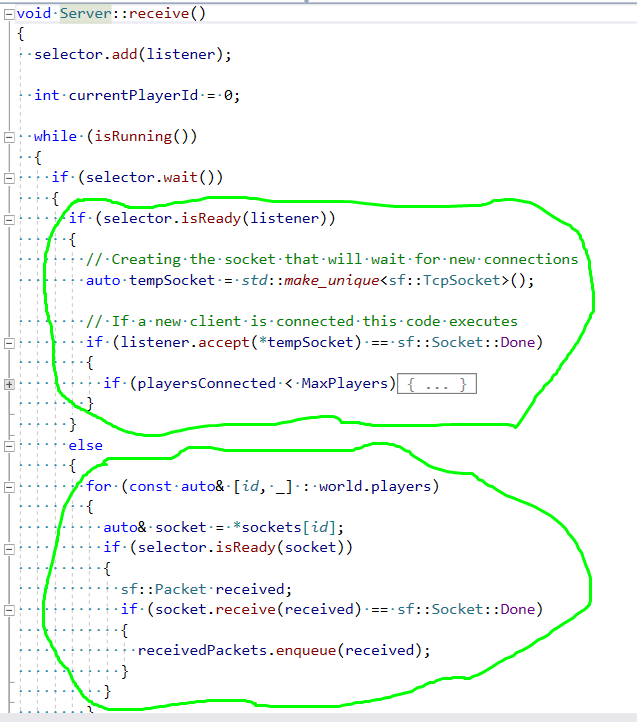


Флажок dirty указывает, что какое-то изменение произошло, требующее оповещения всех клиентов. Нужно, чтобы лишний раз не забивать сеть ненужными сообщениями.

std::lock\_guard нужен для корректной обработки ситуации, когда в момент отправки сообщения синхронизации хочет подключиться новый игрок.

Пакет, который отправляем, начинается с типа сообщения … чтобы клиент понимал, как последующие данные интерпретировать.

Дальше посмотрим на метод **receive**:



В бесконечном цикле два события обрабатываем: сверху – подключение нового игрока … проверяем, что клиентов не слишком много, присваиваем клиенту идентификатор, ставим игрока на случайное место карты и оповещаем клиента, что он зарегистрирован.

Снизу, обрабатываем пришедшие события обновления от клиентов … в данном примере я решил сложить всё их очередь, чтобы если обработка обновлений притормаживает, чтобы это всё копилось в отдельной очереди … на самом деле, думаю, что если вместо этого промежуточного складывания в очередь вставить обработку этих сообщений … что происходит сейчас в методе update, то всё также будет работать нормально, только упростит код.



Update по сути выгребает сообщений из очереди и применяет их к игровому миру.

Вот такие очереди в жизни применяют для целей надежности в качестве промежуточного буфера … вы можете, например, смотреть на количество сообщений в очереди, и если оно начинает расти, то заранее предпринять какие-то меры – начать пропускать менее важные сообщения, обрабатывать сообщения «приблизительно», зато быстро и т.п.

или оповестить о проблеме заранее хотя бы … а не постфактум, когда из-за переполнения важные сообщения потеряются.

Как можете видеть вместо std::queue из стандартной библиотеки, я написал свой класс SafeQueue, чтобы его можно было использовать в многопоточном режиме, когда несколько классов одновременно пишут/читают из очереди.