Масштабируемый движок для нагруженных real-time web приложений на Golang

Ромащенко Андрей

2021

Аннотация

Реализация сетевого взаимодействия это основная задача современного программного обеспечения. Во многих случаях, взаимодействие происходит между небольшим количеством серверов, и большим количеством клиентов. Не редко клиентом выступает браузер. Зачастую, приложение не накладывает серьёзных ограничений на скорость взаимодействия между клиентом и сервером. Так же зачастую логика серверного взаимодействия в достаточной мере отделена от логики клиентного взаимодействия. Для этого случая существуют стандартные решения, которые заключаются в том что клиент может запрашивать данные у сервера по мере необходимости, и в этот же момент передавать свои. Однако - существует значительная часть приложений, для которых этот способ не подходит. Например IO-игры, в которых клиентам требуется получать информацию как можно скорее. Так же играм необходимо уметь предсказывать развитие игровых событий до получения свежих данных.В классическом подходе для этого потребовалось бы дублировать логику сервера на клиенте.

В последние несколько лет стало возможно обойти эти сложности, используя технологии *webassembley* и *websocket*. В этой работе будет рассмотрен подход к написанию движка для подобных проектов, описаны необходимые примитивы и реализован сам движок на языке программирования Go.

Оглавление

Введение

1.1 Основные понятия

Введём основные понятия, которые будут использоваться в дальнейшем.

Сеть — несколько компьютеров, соединённых между собой физическими средствами связи, имеющие возможность обмениваться обобщениями при помощи сетевых протоколов.

Сетевое соединение — абстракция над сетевой инфраструктурой и сетевыми протоколами, позволяющая компьютерным программам обмениваться сообщениями.

Сервер — компьютерная программа, подключённая к сети, обрабатывающая входящие сетевые соединения. В рамках данной работы, рассматриваются сервера реализующие логику приложения, и позволяющие обмениваться информацией соединённым клиентам.

Клиент — компьютерная программа, при запуске которой инициируется соединение с сервером и начинается обмен сообщениями. В рамках данной работы, клиент запускается в браузере, клиент реализует логику по получению пользовательского ввода и вывода.

Браузер — компьютерная программа, стандартизирующая вид клиентов, реализующая абстракции над вводом и выводом данных пользователя, предоставляющая эти абстракции клиентам.

Сетевой протокол — текстовое описание действий, которые необходимо совершить чтобы реализовать сетевое соединение с заданными свойствами.

Websocket [1] — сетевой протокол, реализующий сетевое взаимодействие. Позволяет клиенту и серверу обмениваться сообщениями. Отправитель может отправить в вебсокет сообщение, а получатель его получает, отправителем может выступать как клиент так и сервер. В отличии от классического протокола TCP поддержан в браузере, является одним способом реализовать соединение заключающееся в обмене сообщений.

Webassembley [2] — формат хранения компьютерных программ. Современные браузеры способны исполнять программы в формате webassembley. Для многих языков программирования реализована возможность конвертировать их программный код в webassembley, что позволяет запускать эти программы в браузере.

Real-time программа — в рамках данной работы, программа, имеющая задержку между вводом пользователем данных и выводом реакции на них не более 100 миллисекунд, с допустимыми редкими длительными задержками. (Этот термин значительно отличается от общепринятого определения, заключающегося в том что программа даёт гарантию о времени своего исполнения которая соблюдается во всех случаях).

Приложение — в рамках данной работы, компьютерная программа, симулирующая взаимодействие объектов. Симуляция происходит на сервере, клиенты могут влиять на состояние связанных с ними объектов. Клиенты получают состояние симуляции и выводят его на экран пользователя.

1.2 Цели работы

- Изучить проблемы, возникающие при проектировании сервера для real-time браузерных приложений,

- Исследовать проблемы, возникающие при проектирование клиентской части real-time браузерных приложений,
- Описать и разработать примитивы, позволяющие создавать real-time приложения,
- Реализовать библиотеку для создания конкретных real-time web приложений.

1.3 План работ

Во втором разделе работы будут описаны проблемы, возникающие в создании серверов и клиентов для real-time web приложений, такие как обработка ввода, обработка коллизий, компенсация сетевой задержки, удаление устаревших данных.

В третьем разделе будут предложены конкретные абстракции, позволяющие обобщить описание приложения, разделить общую логику, которая потребуется любому приложению, от логики специфической для данного приложения.

В четвёртом разделе будет описана реализация простой игры, использующая абстракции описанные в разделе три.

Архитектура real-time браузерных игр

2.1 Сервер

В этой секции будет описан ряд проблем, с которыми сталкивается разработчик сервера для real-time web приложения. Первая из них это создание протокола взаимодействия.

2.1.1 Протокол взаимодействия

Протокол взаимодействия между клиентом и сервером в любом приложении зачастую строится на использовании существующих протоколов, поверх которых создаётся новый протокол, решающий конкретную задачу. Для приложений в рамках данной работы от протокола зачастую требуется обладать следующими свойствами:

- В одном соединении участвует один сервер и несколько клиентов
- Клиенты находятся в браузере, что налагает ограничение на возможные сетевые протоколы
- Клиенты не общаются напрямую между собой: в браузере существует протокол для взаимодействия клиентов между собой в обход сервера [3] однако

зачастую нет выгоды посылать сообщения между клиентами в обход сервера, так как серверу в любом случае нужно пересылать сообщения о не подконтрольных клиентам сущностях отдельно.

- Клиентам необходимо иметь возможность запрашивать статические данные у сервера, данные общие для всех клиентов, то есть имеет смысл не получать данные для каждого клиента а дать возможность клиентам самим обращаться к этим данным.

- Чаще всего клиент генерирует событие - ввод пользователя, отправляет его серверу. Сам же клиент зачастую дожидается поступления обновлений от сервера о происходящем для синхронизации.

Важной особенностью подобного протокола является требования быстродействия. Допустимая задержка ввода данных пользователем может быть порядка 100-500 миллисекунд [4]. Допустимы редкие потери объектов, если при этом достигается достаточно большая общая скорость взаимодействия.

В следующей части будет предложен алгоритм, соответствующий указанным требованиям.

2.1.2 Сессии и объекты

Зачастую в приложении требуется поддержать возможность нескольких одновременно запущенных сессий. Даже если приложение подразумевает соединение всех клиентов на едином пространстве, может потребоваться дополнительная сессия-меню, куда клиенты попадают перед началом игры. Соответственно естественно возникает понятие игровой комнаты, как абстракции над сессией, обладающей некоторыми свойствами:

- Комната содержит в себе объекты и занимается их обработкой

- В рамках обработки, комната перемещает объекты, проверяет их столкновения, удаляет необходимые, собирает информацию об сущностях, изменивших своё положение, информацию о которых нужно отправить клиентам

- Было бы удобно хранить комнату как на клиенте так и на сервере, чтобы иметь возможность запускать симуляцию комнаты на клиенте, не дожидаясь ответа от сервера.

Комната сама по себе не содержит в себе какого-либо состояния, целиком полагаясь на объекты внутри. Однако, для удобства использования, имеет смысл создать некоторое внутреннее состояние, чтобы наперёд заданная функция могла его получать и изменять состояние комнаты соответственно.

Так же у комнаты есть требование по быстродействию, нужно уметь быстро добавлять и удалять элементы, находить элемент по номеру. Соответственно, предполагается хранение элементов комнаты в хештаблице. В таком виде возможно реализовать все действия происходящие с комнатой на каждом шаге симуляции как $O(n)$ от количества объектов в комнате, кроме обработки столкновений, ведь наивный алгоритм проверки столкновений предполагает асимптотику $O(n^2)$ то есть проверку каждого объекта с каждым. В разделе три мы предложим решение, как оптимизировать данную проблему тем самым увеличив максимально поддерживаемое количество одновременно существующих сущностей.

2.1.3 Хранение данных и сборка мусора

В предыдущей секции мы определили понятие Комнаты, в рамках которой происходит действия. В данном разделе сформулируем что может происходить с комнатами в рамках приложения:

- Комната создаётся программным кодом, в момент подключения нового клиента или же в момент исполнения другой комнаты

- Во время исполнения комната периодически сохраняет своё состояние в хранилище, чтобы не утерять его при остановке сервера

- Комната исполняется, пока есть подключённые к ней клиенты. Как только последний клиент покидает комнату, остаётся несколько вариантов действий

- Комната может быть удалена, чтобы не засорять оперативную память компьютера

- Комната может продолжать исполнение, ожидая скорого появления клиентов (актуально для комнат-лобби)

- Комната сохраняется в базу данных, с возможностью будущего продолжения работы

- Сервер запускает приостановленную комнату когда в неё подключается новый клиент

Соответственно возникает необходимость хранить комнаты не только в оперативной памяти но ещё и на диске. В следующем разделе будет подробнее описано подобное хранилище.

2.2 Клиент

В этой секции мы подробнее остановимся на проблемах, возникающих при работе клиента. В рамках нашего сетевого протокола, клиент обрабатывает тот же самый объект комнаты что и сервер, однако он использует методы объектов связанные с отрисовкой и получения ввода (в случае объектов, которыми управляет клиент)

2.2.1 Отрисовка происходящего

Выше мы описывали требования от комнаты как хранилища объектов с точки зрения контейнера, в который могут добавлять и удалять значения, а так же с точки зрения контейнера, разделённого на квадраты-зоны. Сформулируем требования к комнате с точки зрения отрисовки объектов:

- Для каждого объекта (подконтрольного клиенту) хотелось бы уметь находить ближайшие объекты, чтобы отобразить только их, и не вызывать функцию отрисовки лишний раз. Отрисовка объекта не попадающего в поле зрения происходит быстро, однако пропадает необходимость итерироваться по всем объектам для отрисовки одного кадра.

- Для изображения необходима виртуальная камера, фактически камера должна строить изображение на основе прямоугольника, описывающего ту часть игрового поля которое попадает в объектив. Соответственно этот игровой прямоугольник камера должна привести к размерам прямоугольника - окна браузера.

- Необходимо поддерживать некий порядок обода объектов при отрисовке, чтобы избегать проблему "накладывания"одного объекта на другой. Так же хотелось бы иметь возможность хранить элементы некоторыми "слоями чтобы иметь возможность сперва отрисовать заднюю часть изображения потом переднюю. Таких слоев может быть несколько.

Для отрисовки мы во многом полагаемся на встроенный в браузер инструмент - canvas [5]. Но для эффективной отрисовки при помощи слоёв требуется поддержка правильного обхода объектов при их отрисовке. Так же необходимо на стороне клиента хранить полученные от сервера текстуры, перед передачей их в canvas для отрисовки.

2.2.2 Получение пользовательского ввода

В начале следующего раздела описывался протокол взаимодействия клиента с сервером, соответственно как только клиент получает объект, которым клиент управляет, клиент начинает запрашивать от пользователя ввод. Здесь опять же, используются встроенные возможности браузера, как-то определение нажатой клавиши или получение положения курсора мыши.

Каждый управляемый пользователем объект имеет дополнительные возможности, а именно он может создать объект на основании введённых пользователем данных, и по переданному объекту соответствующе изменить своё поведение. Соответственно клиент запрашивает подобные изменения у объекта с некоторой периодичностью и передаёт их на сервер. Локально он их применяет сразу же, чтобы уменьшить ощущаемую задержку ввода. Зачастую события о перемещении принимаются сервером, так что имеет смысл их обработать заранее, не дожидаясь ответа.

2.2.3 Компенсация сетевой задержки

В приложении так или иначе возникает задержка ввода. Задержка ввода складывается из трёх основных компонент:

- Сетевая задержка - время между отправкой клиентом данных на сервер и получением ответа от сервера. На эту задержку сложно повлиять, её значения варьируются в широком пределе.(может достигать сотен миллисекунд)
- Задержка цикла с логикой - время которое тратится сервером на обновление сущностей - при правильной реализации не зависит от количества сущностей, почти не варьируется, в целом не велика. (порядка десятка миллисекунд)
- Задержка отрисовки - время занимаемое на отрисовку нового кадра. При

правильной реализации не варьируется, однако его значительно не уменьшить. (порядка десятка миллисекунд)

Соответственно, можно увидеть, что много времени занимает именно сеть, данные пересылаются между клиентом и сервером. Однако, действуя в предположении, что мы на клиенте знаем почти то же самое что и сервер, мы можем запустить интерполяцию действий, для этого достаточно запустить обновление объектов на клиенте, а не только на сервере. Когда сервер наконец-то ответит мы просто обновим соответствующий объект. Таким образом можно скомпенсировать часть задержки, однако если задержка будет слишком велика то возможны визуальные "скачки"объектов.

Описание, исполнение, синхронизация приложений

3.0.1 Протокол приложения основанный на Websocket

Во втором разделе в первой части были приведены требования которым должен соответствовать протокол обмена данными между клиентами и сервером. В рамках данной работы предлагается следующий алгоритм:

- Сервер по запросам умеет предоставлять файлы, при помощи протокола HTTP, в частности файл *index.html*, внутри которого находится программный код клиента

- В момент запуска в браузере клиент инициирует соединение с сервером при помощи протокола Websocket

- Сервер при создании нового соединения сперва отправляет клиенту через Websocket информацию о текущих объектах. Эта информация имеет малый объём, а так же постоянно изменяется, что не позволяет раздавать её статически всем клиентам.

- Затем сервер выбирает, какой сущностью должен управлять клиент и отдаёт ему её идентификатор внутри множества текущих объектов.

- Сервер принимает от клиента сообщения о пользовательском вводе асинхронно, клиент не ждёт мгновенного ответа от сервера, это снимает нагрузку с сервера.

- Сервер исполняет необходимую логику, и если в состоянии каких-то объектов произошли изменения то посылает эти изменения через Websocket всем активным клиентам.

- Клиенты сохраняют обновления от сервера в свою локальную копию множества объектов, соответственно не нужно каждый раз посылать информацию обо всех объектах.

- Клиент может обращаться к статическим файлам на сервере, таким как текстуры, аудио фрагменты.

Таким образом, мы используем преимущества Websocket заключающееся в переиспользовании TCP соединений, что позволяет снизить количество пересылаемых объектов по сети [6].

В тех местах где данные не изменяются с течением времени (например текстуры, которые изменяются только при обновлении приложения) имеет смысл использовать классический подход с HTTP, по скольку в таком случае возможно кеширование данных на стороне браузера и соответственно уменьшение объёма пересылаемых данных [7].

3.0.2 Игровая комната и сущности

В предыдущем разделе были сформулированы требования по быстродействию для комнаты. Комнату можно легко реализовать наивным способом используя хештаблицу для хранения объектов.

Можно разделить объекты на категории, хранить отдельно хештаблицы движущихся объектов и хештаблицу объектов которые могут сталкиваться. Это

позволяет ускорить цикл обработки который занимает большую часть времени работы комнаты.

Так же в предыдущем разделе была сформулирована проблема рассчета коллизий. В игре необходимо уметь быстро находить объекты, физически находящиеся рядом с заданным, делать это быстрее чем за $O(n)$. Для этого самым простым способом будет использовать систему квадратов, каждый из которых имеет фиксированные размеры. В таком случае, зная местоположение можно понять в каком квадрате он находится (просто разделив координаты объекта на размеры квадрата), затем высчитать функцию столкновения только с теми объектами, которые находятся в тех же квадратах что и данный. Важно хранить объект во всех квадратах, куда он попадает, иначе можно пропустить столкновения на границе квадратов.

Цикл обновления комнаты выглядит так:

- Комната обновляет все объекты которые могут передвигаться
- Для этого комната передаёт время, произошедшее в симулируемом мире с прошлого вызова функции
- Комната рассчитывает столкновения объектов, которые могут сталкиваться. Для этого использует дополнительную структуру данных на объектах
- В случае сервера, комната обрабатывает входящие игровые события и посылает исходящие
- Цикл обновления запускается раз в несколько миллисекунд чтобы не загружать процессор

В момент обработки коллизий игра не рассчитывает явно, столкнулись ли объекты или нет, каждый объект сам проверяет, столкнулся ли он с переданным объектом или нет. Это позволяет реализовывать более гибкие системы, однако функция проверки столкновений так же реализована. На данный момент, каждый объект может иметь вид объединения ориентированных прямоугольников и

кругов. Соответственно, при рассчитывании столкновений двух объектов проверяются пересекаются ли данные множества, а так же предоставляется вектор, двигаясь по которому объекты выйдут из столкновения. В таком виде остаётся опасность туннелирования, однако для этого объект должен двигаться с очень большой скоростью, чтобы "проскочить"другой объект.

3.0.3 Событие

В предыдущих разделах описывался механизм создания сетевого протокола. В рамках него, несколько клиентов общаются с сервером. В этой секции будет описано какими именно сообщениями обменивается клиент и сервер.

Все сообщения структурированы и имеют вид "игровых событий". Игровое событие это объект, у которого есть тип а так же идентификатор отправителя. У события так же есть дополнительное поле, которое используется в зависимости от типа. Существует несколько заранее заданных типов событий, такие как получение пользовательского ввода, обновление игровой сущности, событие в котором описана вся комната целиком, событие удаление объекта, событие окончание симуляции для данного клиента.

События от клиентов попадая на сервер выстраиваются в очередь событий, события попадающие к клиенту применяются сразу же. Заданные типы событий обрабатываются сразу же, а для пользовательских типов событий вызывается соответствующий обработчик, если такой найден.

Помимо клиента и сервера события могут быть сгенерированы объектами во время их обновления, для этого в функцию обновления комната передаёт объект для передачи сообщений.

События, так же, имеют возможность преобразовываться в json и наоборот, таким образом они передаются по сети. Событие может содержать внутри себя игровой объект, по этому от объекта требуется возможность преобразовать себя в набор байт, и быть впоследствии распакованным из набора байт.

3.0.4 Графический контекст

Для взаимодействия с предоставляемыми браузером возможностями отрисовки, написана прослойка, так называемый графический контекст. Графический контекст это специальный объект, создающийся во время создания клиента. Формат Webassembley позволяет запускать программу на произвольном языке программирования из браузера, однако помимо этого он позволяет обращаться к возможностям браузера используя вызовы Javascript. Соответственно контекст создаёт внутри себя контекст из canvas, и каждый кадр передаёт туда свежее изображение. Однако с целью оптимизации, чтобы ликвидировать затраты на вызовы javascript изображение создаётся целиком на стороне Go, и зачем направляется в javascript.

Помимо работы с canvas, внутри графического контекста поддержана возможность создавать html элементы в качестве отрисованного результата для объекта. Это позволяет создавать поля ввода используя средства html вместо создания нового объекта.

Графический контекст предоставляет интерфейс, позволяющий отрисовать либо произвольную геометрическую форму либо заданную текстуру в заданной точке. Когда все объекты отрисованы, текстура отправляется в canvas.

Графический контекст требует, чтоб сперва были отрисованы объекты, которые находятся на заднем плане, а затем объекты на переднем плане. Эта возможность нужна не всегда, если объект не поддерживает слои, то он рисуется всегда на самом переднем плане. Для реализации этой возможности, в объекте Комнате поддержана ещё одна структура данных, массив хештаблиц, каждый массив обозначает слой со своим номером, от самых дальних до самых ближних. В хештаблицах хранятся указатели на объекты. Это позволяет быстро обходить все объекты в правильном порядке, а так же быстро добавлять и удалять объекты.

3.0.5 Хранилище данных комнат

Во втором разделе описывалась проблема хранения комнат, пустую комнату нужно удалять или хранить или продолжать выполнять в зависимости от ситуации. Для этого пользователю передано хранилище данных комнат. Когда комнату покидает последний клиент вызывается либо переданная пользователем функция, либо комната помечается в хранилище как удалённая. Хранилище представляет собой объект, с хештаблицей, где хранятся активные исполняемые комнаты и подключением к базе данных. По скольку комнаты сами по себе представляют сериализуемый объект, они спокойно могут храниться в любом возможном хранилище. Go предоставляет абстракцию над базами данных, так что хранилище можно выбирать самостоятельно.

Так же хранилище предоставляет возможность перемещать сущность из комнаты в комнату. При этом в одном режиме сущность просто перемещается, во втором режиме данному подключённому клиенту назначается сущность с переданным номером в новой комнате.

При создании, хранилище запускает процесс, который раз в некоторое время (раз в несколько минут) сохраняет все активные не динамические комнаты в базу данных, на случай поломок. На данный момент, нет возможности переподключиться к уже запущенной комнате (движок никак не использует cookie браузера), однако упавшая комната будет поднята когда к ней кто-то обратится.

Реализация игры с помощью движка

В этом разделе будут описан пример приложения, которое можно создать при помощи данного фреймворка. Приложение носит демонстрационный характер. Приложение основано на нескольких комнатах. В начале игрок попадает в комнату - лобби. Там игрок видит приветственную фразу, находясь там он дожидается появления в комнате ещё одного игрока. Как только это происходит, начинается собственно игровой процесс. Игроки управляют объектами - сферами, каждая сфера может отрастить хвост из сфер по-меньше. Цель игры - добиться того чтобы игрок - противник врезался во второстепенную сферу, при этом не врезаться во второстепенную сферу врага самому. В конце игры игроки попадают в комнату для победы и в комнату для поражения. Затем каждый игрок может переместиться в начальную комнату и начать процесс заново.

Соответственно, в этом приложении существует три наперёд заданные постоянно исполняющиеся, восстанавливающиеся из базы данных при падении комнаты: комната-лобби, комната с победным сообщением, комната с сообщением о проигрыше. Так же - создаются динамические комнаты, они не сохранятся в базу данных при падении.

4.0.1 Создание своих сущностей

Каждая сущность должна удовлетворять интерфейсу, чтобы быть принятой игрой:

```
type Element interface {
  Draw(c canvas.Canvas) // all actions to draw the object
  GetID() int           // for cross element actions
  GetType() int         // for cross element actions
  GetState() ([]byte, error)  // additional method, implement if required
  SetState([]byte) error // additional method, implement if required
}
```

Листинг 4.1: Интерфейс сущности

При помощи этого интерфейса можно создать статическую сущность, она будет находиться всегда в одном месте, она не будет участвовать в взаимодействиях, ею нельзя будет управлять и она никак не обновляется. Такая сущность полезна для создания статических переменных окружения, например задника или картинки:

```
type StaticBackground struct {
  Where     math.Box
  TextureID string
  ID        int
}

func (s *StaticBackground) Draw(c canvas.Canvas) {
  c.DrawShape(s.TextureID, s.Where, math.Box{Size: s.Where.Size})
}
```

Листинг 4.2: Реализация статического окружения

Этот обьект используется в лобби и других статичных комнатах для создания задника при помощи текстуры.

Если пользователь хочет, чтобы реализованные объекты могли обновляться, участвовать в коллизии, быть управляемым игроком, то ему следует реали-

зовать следующие методы:

```
type Playable interface {
  Element
  Input() ([]byte, error) // result used in "input" event
  SetInput([]byte) error // parameter passed from "input" event
}

type Collidable interface {
  Element
  Collide(other Collidable) error // collision should be checked inside
  Collider() math.Shape           // useful in Collide
}

type Movable interface {
  Element
  Move(duration time.Duration, processor EventProcessor) error // all
    changes in object
}
```

Листинг 4.3: Реализация статического окружения

Не обязательно реализовывать все три, например для лобби используется объект, реализующий интерфейс игрока, чтобы клиент мог взять управление, но при этом не делающий ничего, игрок просто дожидается появление нового игрока, затем игра переправляет обоих игроков в "игровую"комнату.

Единственный, реализующий все функции объект это собственно змейка. Змейка состоит из звеньев, каждое звено это окружность, так же она хранит разные дополнительные параметры:

```
type Snake struct {
  Orbs []math.Sphere
  ID   int

  Vel  math.Vector
  Dead bool

  I           PlayerInput
```

```
  Lost        bool
  LastCreated time.Time
}
```

Листинг 4.4: Класс Snake которым управляют игроки

При отрисовке, змейка по очереди рисует все свои звенья. При получении пользовательского ввода змейка просто возвращает системные значения (они прописываются при помощи коллбеков вызываемых в javascript)

```
func (s *Snake) Draw(c canvas.Canvas) {
  for i, o := range s.Orbs {
    if i == 0 {
      c.DrawColor(color.RGBA{G: 255, A: 255}, o, o)
    } else {
      c.DrawColor(color.RGBA{R: 255, A: 255}, o, o)
    }
  }
}

func (s *Snake) Input() ([]byte, error) {
  s.I.Target = input.MousePosition
  //o.I.Moving = input.MousePressed
  s.I.Moving = true
  s.I.GenNew = input.Pressed[input.KEY_SPACE]
  return json.Marshal(s.I)
}


func (s *Snake) Collide(other elements.Collidable) error {
  info := math.Collide(s.Orbs[0], other.Collider())
  if info.Collided {
    s.Dead = true
  }
  return nil
}
```

Листинг 4.5: Функции передвижения получения ввода обработки столкновений

При столкновении змейка использует библиотечную функцию проверки столкновений, и при наличии столкновения погибает.

Наконец, в функции изменения положения змейка либо перемещается, либо отправляет сообщение о проигрыше.

```
func (s *Snake) Move(duration time.Duration, processor elements.
    EventProcessor) error {
  playerOrb := s.Orbs[0]

  if s.Dead && !s.Lost {
    s.Lost = true
    return processor.ProcessEvent(event.Event{Type: "lose", From: s.GetID()
    })
  }
  if s.I.GenNew && time.Since(s.LastCreated) > time.Second {
    // creating new orb, behind the last one
  }
  if s.I.Moving {
    // moving code ommited, every next orb move towards previous
  }
  return nil
}
```

Листинг 4.6: Функция изменения состояния змейки

В случае смерти, змейка посылает особое событие ”*lose*” обработчик для которого задаётся в следующей части.

Перед использованием объект необходимо зарегистрировать, это необходимо чтобы правильно распаковывать созданный пользователем объект при пересылке по сети.

```
func init() {
  elements.GenElements[SnakeType] = func() elements.Element {
    return &Snake{}
  }
}
```

Листинг 4.7: Функция регистрации типа объекта

4.0.2 Расширение возможностей комнаты

TODO: Добавить сюда снипеты с кодом особых комнат, функций обработчиков, с пояснениями

4.0.3 Сборка и запуск

TODO: Клиент-сервер, клиент нужно собрать в wasm, при этом нужно чтобы system/js не импортировался сервером (тогда он не соберется, system/js используется только в wasm), про то что игра начинается в index.html, про то что серверу нужно передать конфиг с конфигом базы данных, и т. д.

Выводы

В данной работе был исследован ряд проблем связанных с разработкой real-time web приложений, сложности возникающие при создании сервера, клиента, предпочтительные протоколы и алгоритмы. Сложности обработки столкновений и хранения объектов. Результатом работы стал движок, который предоставляет пользователю возможность описать объекты, сконцентрировавшись только на их поведенческой логике. Движок предоставляет сервер, к которому могут подключаться клиенты из браузера. Благодаря системе абстракций "Комната"есть возможность обрабатывать несколько игр параллельно, есть возможность сохранить состояние приложения и продолжить его исполнение. Полученный исходный код выложен на GitHub.

Однако исследование не завершено, главным образом из за возможных оптимизаций существующего кода. Главное необходимое улучшение - добавление возможности переподключения к внезапно остановленной комнате. Так же движку пока что не хватает геометрических возможностей, отсутствую геометрические формы отличные от сферы и ориентированного прямоугольника. Так же отсутствует встроенная физическая модель. которая однако требуется в большинстве проектов. Так же нет готовой поддержки анимаций - что требуется ещё чаще. Сюда же можно отнести эффекты постобработки. Всё это может быть постепенно добавлено в существующую кодовую базу без существенных её изменений.

Литература

[1] RFC. Websocket protocol. "https://datatracker.ietf.org/doc/html/rfc6455".

[2] Webassembley. "https://webassembly.org/".

[3] Webrtc. "https://webrtc.org/".

[4] Cong Ly. Latency reduction in online multiplayer games using detour routing.

[5] Canvas. "https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API".

[6] Andrew Lombardi. *WebSocket: Lightweight Client-Server Communications*.

[7] Websocket or http. "https://developerinsider.co/difference-between-http-and-http-2-0-websocket/".