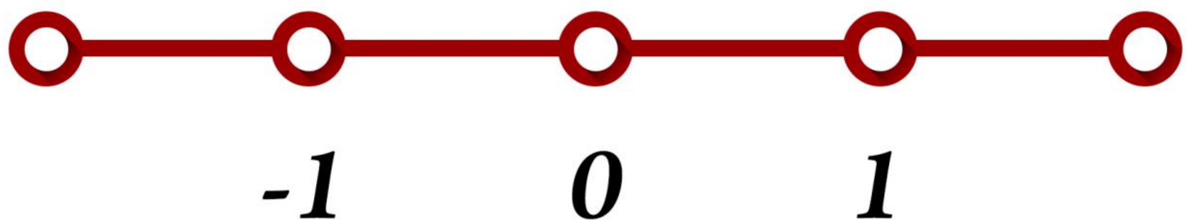
ХРОНОЛОГИЧЕСКАЯ МОДЕЛЬ ПРОГРАММИРОВАНИЯ



-1 0 1

упрощение программирования лагокомпенсации в сетевых играх

Cheryl Savery

T. C. Nicholas Graham

1. ВВЕДЕНИЕ

Алгоритмы лагокомпенсации необходимы в рамках программирования сетевых игр, когда дело касается работы со временем и распределенными игровыми параметрами и состояниями. Сложность реализации этих алгоритмов ограничивает возможности проведения экспериментов для нахождения и тестирования наиболее подходящего алгоритма для конкретной ситуации, для определения, насколько тот или иной алгоритм был бы эффективен. Решение этого вопроса заключается в нахождении модели программирования, обеспечивающей работу со временем. В данной статье описывается подобная модель программирования, хронологическая модель. Хронологии значительно сокращают время, требуемое на реализацию технологий лагокомпенсации, позволяя явно обращаться ко времени. Хронологическая модель легла в основу фреймворка Janus Toolkit.

Для обеспечения лагокомпенсации необходимо хранить не только текущие значения свойств различных игровых объектов (распределенных данных в сетевых играх), но и данные о том, когда эти свойства были изменены. Нынешние языки программирования не предоставляют средств для работы со временем. Мы исправим этот недостаток, представив собственную хронологическую модель программирования для разработки сетевых игр. Хронологии - это новая модель программирования, обеспечивающая доступ к распределенным данным по указанному времени. Она позволяет программисту получить необходимые значения переменных в предыдущих (прошлых) и последующих (будущих) состояниях.

Хронологические переменные могут быть распределены между игроками, позволяют программисту изменять скорость течения игрового времени и создавать отличающиеся (расходящиеся) хронологии для каждого из игроков. Делая время неотъемлемой частью модели программирования, хронологии упрощают реализацию алгоритмов лагокомпенсации. Это, в свою очередь, позволяет экспериментировать с алгоритмами - можно использовать разные сочетания алгоритмов и создавать новые жанры многопользовательских игр.

Наша хронологическая модель была применена в Janus toolkit и используется авторами и другими программистами для разработки алгоритмов лагокомпенсации и для создания многопользовательских игр. Для демонстрации возможностей хронологической модели мы рассмотрим

реализации нескольких алгоритмов, а также используем эту модель для расширения алгоритма Локальных фильтров восприятия, соединив его с Плавными исправлениями, предоставив новое решение для проблем связанных с игровой физикой.

Статья организована следующим образом. Мы начнем с рассуждений о том, что время играет важную роль в лагокомпенсации, что мотивирует нас к использованию хронологической модели программирования. Далее мы представим нашу хронологическую модель, покажем, как она облегчает реализацию ряда алгоритмов лагокомпенсации, включая наше новое расширение алгоритма Локальных фильтров восприятия. Затем мы обсудим реализацию этой модели в Janus toolkit и расскажем о принципах её применения.

2. ПОЧЕМУ ВРЕМЯ ТАК ВАЖНО

Технологии лагокомпенсации используются для того, чтобы сократить неожиданности в поведении сетевой игры при возникновении задержек в передаче данных. К примеру, неожиданное поведение (лаг) проявляется как непопадание при стрельбе во врага при явном визуальном попадании снаряда в цель, прерывистое движение, скачки, столкновение с невидимыми препятствиями, или же попадание персонажей игроков в те места, куда они не могут попасть. Эти проблемы появляются при различии в визуальном восприятии игроками одних и тех же общих игровых данных из-за несвоевременной передачи изменений этих данных по сети.

Алгоритмы лагокомпенсации направлены на сокращение таких неожиданностей за счет смягчения эффекта от сетевых задержек. Если говорить детально, существует три основных подхода к уменьшению лагов:

1. Расчет пути

2. Отложенный локальный ввод

3. Удаленная задержка

Эти три подхода сложны для программирования, а их эффект сложно проанализировать. Исходя из этих трудностей, можно утверждать, что программисту необходимо постоянно иметь дело со временем и с состоянием общих (распределенных) данных. Например, алгоритм лагокомпенсации

должен учитывать, когда и где был произведен выстрел, направление этого выстрела, и кроме того, должен исключить ситуации, когда уже умерший персонаж не может стрелять. Далее мы опишем 4 ключевые проблемы, освещающие тему значимости времени изменения общих данных в сетевых играх.

Проблема устаревших сообщений

Поскольку на передачу сообщений по сети затрачивается время, обновления состояний обязательно устаревают в момент их прибытия. Клиенты должны иметь доступ к информации, корректирующей рассинхронизацию данных. Например, сообщение о том, что персонаж другого игрока передвинулся, на деле информирует игрока о том, где другой игрок находился некоторое время назад. Из-за того, что время передачи данных по сети постоянно меняется, бывает сложно оценить, когда именно сообщения были отправлены (иначе говоря, насколько устарели пришедшие данные). Решение этой проблемы заключается в присвоении сообщениям временно'й метки, по которой можно определить разницу во времени между игроками. Таким образом, проблема устаревших сообщений решается необходимостью для программиста рассматривать сообщения не как текущие значения, а как значения, описывающие состояния в предыдущие моменты времени.

Проблема устаревших состояний

Некоторые игровые состояния обновляются часто. Например, в играх где изобаржение на экране обновляется 60 кадров в секунду, позиция персонажа игрока может обновляться каждые 17мс. Для сохранения пропускной способности, игры обычно передают данные о текущем состоянии с меньшей частотой, например каждые 50мс. Обновления могут быть отложены, если клиент игрока может узнать, что другие клиенты имеют достаточно информации для точного прогнозирования обновленного состояния. Клиенты, соответственно, нуждаются в визуализации нескольких кадров, прежде чем поступит новая информация о состоянии, а также они должны иметь возможность оценить текущее состояние другого клиента на основе последней доступной информации о состоянии. Это требует хранения информации о времени изменения состояний другого клиента и наличия у них механизмов для компенсации обновлений этого состояния.

Проблема точки зрения

В некоторых играх осознанно используются разные точки зрения для каждого из клиентов. Например, в Half-Life применяется постоянная задержка при движении других игроков. Это повышает предсказуемость движения персонажей за счет увеличения расхождения между состояниями игрового мира у разных игроков. Таким образом, как показано на рисунке 1, вместо использования одного глобального состояния игрового мира, каждый игрок имеет свое личное представление (точку зрения). Игровые действия основаны на том, что игрок видит на своем экране, вместо того чтобы основываться на истинной позиции других игроков. Это позволяет игрокам целиться и стрелять в целевого игрока без проблем с неточными позициями, вызванных задержками сети. Для обеспечения последовательности и избежания читерства, сервер должен принимать во внимание представление каждого игрока и восстанавливать это представление у себя. Серверу требуется отмотать время и воссоздать состояние как целевого так и стреляющего игроков в точке времени когда произошел выстрел.

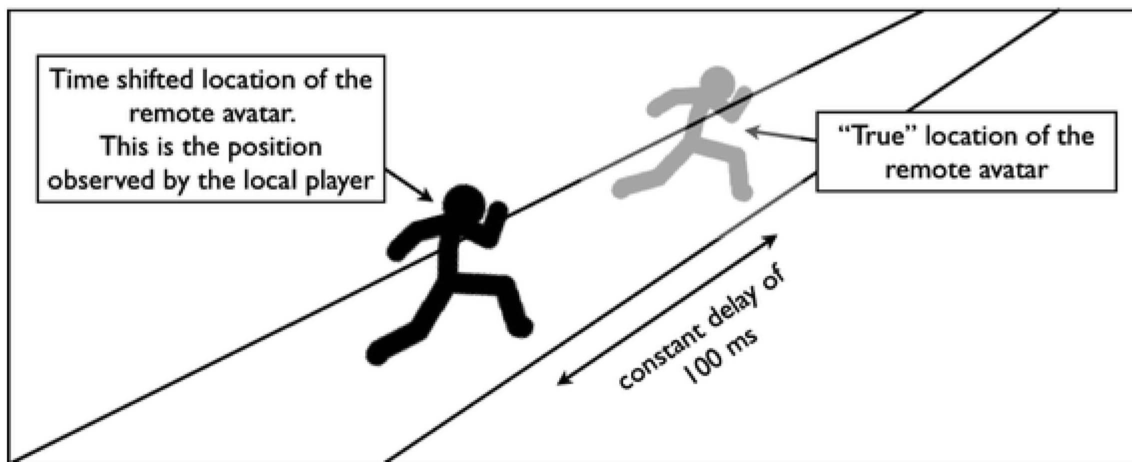


Рис.1

Используя удаленную задержку, каждый игрок имеет собственную точку зрения игрового мира. Представление удаленных персонажей задерживается на фиксированное время, которое больше, чем сетевая задержка. Это позволяет игрокам прицеливаться туда, где они видят персонажа другого игрока, вместо попытки прицеливания в настоящую текущую позицию персонажа.

Проблема множества моментов в одном

Некоторые технологии лагокомпенсации требуют чтобы клиент имел доступ к общему игровому состоянию в прошедшем, текущем и будущем времени.

Например, эта проблема возникает в алгоритмах Плавного исправления, как показано на рисунке 2. Когда клиенты используют прогнозирование (такое как Расчет пути) для экстраполяции позиции удаленной сущности, этот прогноз будет неправильным всякий раз, когда сущность изменит скорость или направление. Как только обнаружена ошибка, вместо немедленного смещения удаленной сущности в правильную позицию, алгоритм Плавного исправления постепенно двигает объект в новую позицию. Это позволяет скорректировать состояние объекта менее резко. Для реализации Плавного исправления, программист должен иметь доступ к предыдущему состоянию и знать где была сущность, иметь доступ к текущему состоянию, чтобы понять где сущность должна быть и рассчитать будущее состояние для того чтобы со временем плавно объединить эти два состояния.

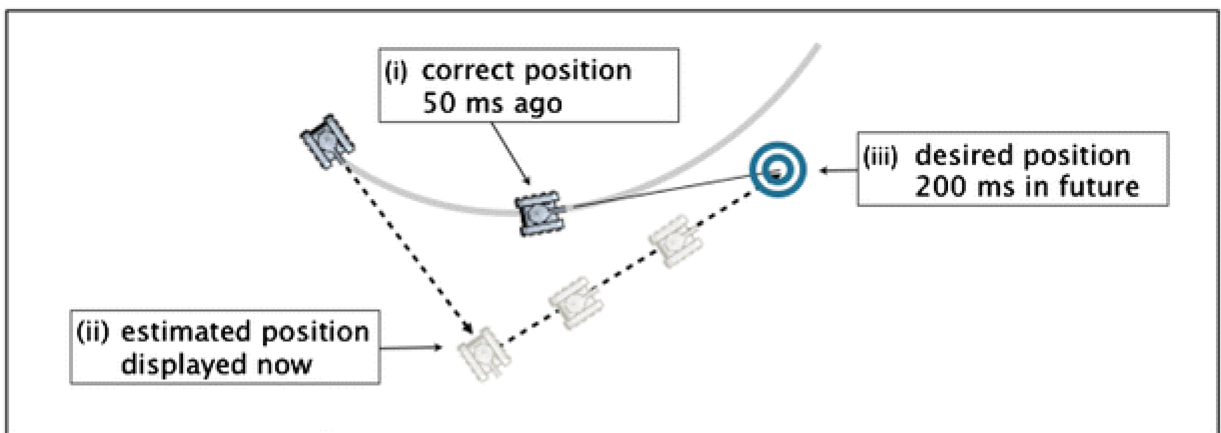


Рис. 2

Плавные исправления требуют доступа к данным о позициях персонажа (i) в прошлом (как указано в последнем сообщении обновления), (ii) на данный момент, и (iii) где бы мы хотели видеть персонажа в будущем.

Как мы видим из этих примеров, работа со временем является основным компонентом алгоритмов лагокомпенсации. Примеры показали необходимость игрокам интерпретировать состояние и обновление состояния относительно прошлого времени и необходимость сервера восстанавливать различия точек зрения игроков, и иногда необходимость игрокам иметь доступ к значениям в прошлом, нынешнем и будущем состояниях.

Традиционные модели программирования рассматривают состояние как единичное значение, существующее в текущий момент времени. Это первопричина сложности всех алгоритмов лагокомпенсации, в основе которых лежит работа с различиями точек зрения игроков. Мы покажем, что

модель программирования, которая неотменно включает в себя время, значительно упростит реализацию этих алгоритмов.

3. ХРОНОЛОГИЧЕСКАЯ МОДЕЛЬ

В распределенных системах общие данные обычно хранятся в виде одиночных значений, отображающих данные на текущий момент времени. Как мы рассуждали, это приводит к сложностям при программировании алгоритмов, в которых задействовано время. На контрасте наша хронологическая модель представляет общие данные как значения, индексированные по времени. Переменные представлены всеми ранее принимаемыми ими значениями, и всеми значениями, которые они будут принимать в будущем. Хронология предоставляет:

- Операции получения и установки (Get и Set), которые будут давать доступ к хронологическим значениям в указанное время

- Функции интерполяции и экстраполяции для оценки значений в те моменты времени, в которых точное значение не известно.

Как мы увидим, эта простая модель распределенных данных даст широкие возможности для обеспечения лагокомпенсации простыми выражениями.

На рисунке 3 показано как комбинированы элементы хронологической модели. В примере представлены значения v1, v2 и v3 для времени t1, t2 и t3, соответственно. Это известные значения, то есть они были установлены в хронологию.

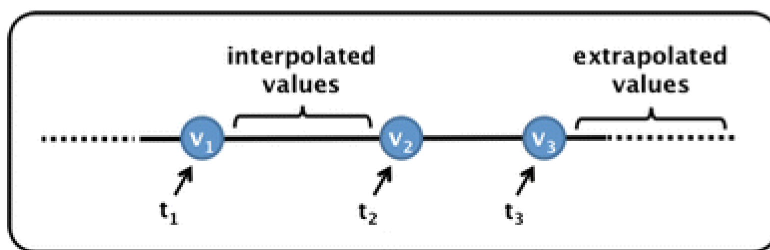


Рис. 3

Элементы хронологической модели: хронология включает набор прошлых и будущих значений (v1, v2, … v3), наряду со значениями времени, когда они были установлены (t1, t2,… t3). Значения, которые находятся между этими значениями, можно получить, используя функцию интерполяции, а значения после последнего заданного - с помощью функции экстраполяции.

Для демонстрации этого рассмотрим параметр fuel - целочисленное хронологическое значение, представляющее количество топлива космического корабля. Хранение данных в хронологии fuel заключается в простой установке его значения в определенные моменты времени. Таким образом

fuel(-100) = 230;

установит значение 230 этому параметру 100 мс назад и

fuel(0) = 200;

установит 200 для хронологии на текущий момент времени. (Временные параметры заданы в мс, могут быть положительными и отрицательными относительно текущего времени.)

Когда мы пытаемся получить значение, неизвестное в определенный момент времени, оно вычисляется на основе ближайших известных значений. Значение хронологии fuel(−50) будет вычислено как 215 с использованием линейной интерполяции между значениями 200 и 230. Значение fuel(100) будет 170. Хотя наша хронология хранит только 2 значения, мы можем узнать значения в любой момент времени прошедшего и будущего. При добавлении новых значений в хронологию, эти интерполируемые и экстраполируемые значения могут измениться.

Хронологическая модель полностью повторяется для каждого из игроков (клиентов) - у каждого из них хранится локальный хронологический объект для каждого из распределенных состояний, в которых он заинтересован. Если два игрока создают экземпляры одной хронологии, эти хронологии автоматически синхронизируются. На рисунке 4 показаны два игрока, каждый со своей копией одной хронологии. Когда второй игрок добавляет новое значение в локальную хронологию, это значение передается по сети и вызывается удаленная функция обновления у первого игрока. Эта функция определяет как обновляются значения, приходящие по сети. По умолчанию функция обновления просто помещает приходящее значение в хронологию у игрока 1. Тем не менее, мы видим, что есть возможность перезаписать эту функцию, например для плавных изменений.

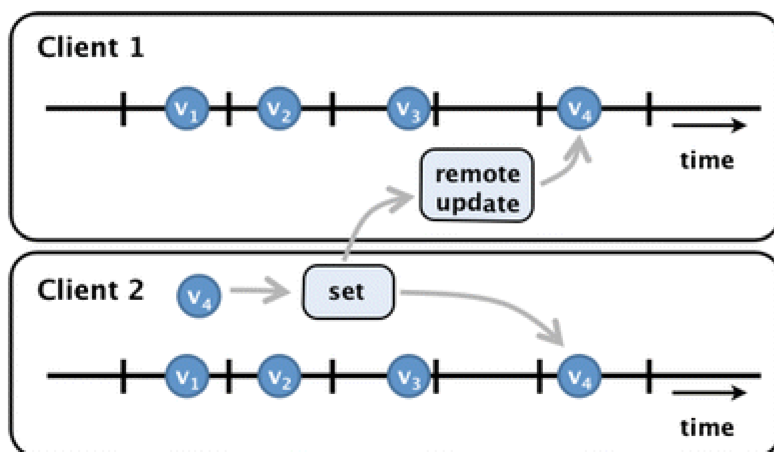


Рис. 4

Хронологии полностью дублируются. Когда один клиент добавляет значение в свою хронологию, это значение добавляется в хронологии всех остальных клиентов.

Для того чтобы продемонстрировать работу хронологической модели, теперь мы обсудим несколько технологий лагокомпенсации, используемых в многопользовательских играх, и покажем, как хронологии могут быть использованы для их реализации.

4. ИСПОЛЬЗОВАНИЕ ХРОНОЛОГИЙ

В данном разделе мы рассмотрим применение хронологий в реализации ряда канонических алгоритмов лагокомпенсации. На примерах мы покажем возможности модели, а также проиллюстрируем её выразительность.

Многопользовательские игры используют разнообразные методы лагокомпенсации. Эти методы, как правило, учитывают пользовательский опыт, а также возможности читерства. Технологии лагокомпенсации обеспечивают компромисс между обеспечением должной степени согласованности и отзывчивостки к командам игрока. Основываясь на этом компромиссе, мы можем разделить методы, используемые для лагокомпенсации на три большие категории: методы прогнозирования, методы отложенного ввода и методы компенсации времени. Алгоритмы для каждой из категорий описаны ниже:

МЕТОДЫ ПРОГНОЗИРОВАНИЯ оценивают текущее значение удаленного состояния из состояния, доступного локально. Например, Расчет пути (dead reckoning) широко используется для оценки текущей позиции персонажей других игроков основываясь на предыдущих позициях и информации о их

скорости, переданных ранее по сети. Это помогает решить проблему Устаревших состояний, описанную ранее, обеспечивая средства для оценки того, что произошло на клиенте другого игрока в промежутке межу моментами обновления состояния. Алгоритмы прогнозирования могут быть использованы и для решения проблем Устаревших обновлений. Когда обновление состояния прибывает из сети, прогнозирование может использоваться для определения текущего значения состояния с учетом времени, когда сообщение было отправлено. Для достижения этого, сообщения обновлений состояния должны иметь временную метку и часы локального и удаленного клиентов должны быть синхронизированны. И наконец, если используется плавная корректировка неточных предсказаний, программисту нужно иметь доступ к состояниям в прошлом, настоящем и будущем, как показано на рисунке 2.

МЕТОДЫ ОТЛОЖЕННОГО ВВОДА, такие как Баскет-синхронизация (bucket synchronization) и Локальная задержка (local lag) откладывают локальные действия для одновременного запуска всеми клиентами. Программирование Локальной задержки требует механизмов отложения ввода и оценки времени доставки сообщения между различными узлами. Это также требует политики обработки сообщений, передача которых занимает больше времени, чем время постоянной задержки. Баскет-синхронизация требует механизма остановки всех клиентов в конкретной точке выполнения, применения всех ожидающих сообщений в определенном порядке и продолжения выполнения.

МЕТОДЫ СМЕЩЕНИЯ ВРЕМЕНИ такие как Удаленная задержка (remote lag) и Локальные фильтры восприятия (local perception filters) вводят в приложение задержку обновления удаленных состояний. Например, как показано на изображении 1, позиция персонажа удаленного игрока задерживается на постоянное число миллисекунд. Этот подход меняет непосредственно методы обновления позиций, чтобы компенсировать скачки, вызываемые различием во времени доставки обновлений по сети.

Все эти методы включают в себя манипуляции со временем - любое прогнозирование будущего, планирование действий для будущего выполнения, или обеспечение расходящихся хронологий для различных игроков. В этом разделе теперь мы покажем, как хронологическая модель упрощает реализацию алгоритмов для каждого из описанных методов.

4.1 МЕТОДЫ ПРОГНОЗИРОВАНИЯ

Методы прогнозирования состоят из двух компонентов. Во-первых, функции экстраполяции используются для вычисления текущего состояния удаленной сущности, основываясь на предыдущих известных состояниях. Например, Расчет пути широко используется для определения позиции персонажей других игроков. Во-вторых, необходим механизм для исправления состояния сущности когда прогнозирование было ошибочным. Дальнейшие подразделы описывают оба компонента более детально и показывают как использование хронологий облегчает их реализацию.

4.1.1 Пример: Расчет пути (dead reckoning)

Расчет пути обычно используется для сокращения количества передаваемых по сети сообщений об обновлении позиции игровых объектов. Алгоритм Расчета пути основан на предположении, что сущности редко меняют направление и скорость, и их предыдущее движение довольно точно описывает их последующее движение. В Стандарте распределенного интерактивного моделирования IEEE определен протокол Расчета пути, согласно которому, для оценки позиции сущности применяется отношение экстраполяции. Вместо передачи сообщений о перемещении всякий раз, когда сущность перемещается, обновления передаются только тогда, когда превышен порог ошибки. Когда Расчет пути используется в многопользовательских играх, игрок, управляющий сущностью, передает данные о положении и скорости другим игрокам, которые используют эту информацию для оценки текущей или будущей позиции сущности.

Алгоритмы Расчета пути, всё же, часто терпят неудачу в связи с задержками передачи данных по сети. Когда игрок получает сообщение обновления позиции, эта позиция устанавливается как позиция сущности в текущий момент времени, а не в момент отправки сообщения. Рассмотрение удаленных обновлений как прошедших событий (т.е. с учетом сетевых задержек) повышает точность Расчета пути. Однако, этот подход требует сложного программирования привязки сообщений ко времени для обновления позиции и синхронизации часов у игроков.

Расчет пути с учетом задержек встроен в хронологическую модель и не требует от программиста дополнительных решений. Функция экстраполяции модели обеспечивает Расчет пути как стандартное действие. Когда обновления получены от удаленного игрока, они автоматически помещаются

в локальную хронологию по времени, когда они были отправлены, а не по времени их получения.

Следующий пример показывает как с помощью хронологии легко реализуется Расчет пути с учетом задержек. Рассмотрим двух персонажей игры - Алису и Боба. Позиция Алисы представлена в хронологии alicePos. Мы предполагаем традиционную игровую архитектуру, в которой пользовательский ввод и вывод изображения на экран происходят асинхронно. Таким образом, когда Алиса перемещается в позицию (x, y), процесс передвижения у игрока Алисы прост:

alicePos(0)=(x, y);

Эта операция помещает в хронологию alicePos значение (x, y) для времени 0 - текущего времени. Затем на клиентах обоих игроков, когда кадр нарисуется, позиция рисования Алисы будет

drawAvatar(alicePos(0));

Таким образом, персонаж нарисуется на данный момент в этой позиции.

У игрока Боба во время прибытия сообщения о передвижении Алисы, значение её позиции автоматически установится в хронологию с учетом задержки сети. Например, если для передачи сообщения, которое содержит позицию Алисы, необходимо 60 мс, то значение автоматически поместится в хронологию на компьютере Боба для времени -60.

Когда клиент Боба использует значение хронологии alicePos(0) для рисования Алисы, оно экстраполируется из последней известной позиции. Этот пример показывает как хронологическая модель обеспечивает Расчет пути с учетом задержек как функцию по умолчанию, без дополнительных доработок.

4.1.2 Пример: Плавные исправления (smooth corrections)

Когда используется Расчет пути и позиция персонажа, полученная из сообщения, сильно отличается от предсказанной, простейшее решение - немедленно переместить сущность в новую позицию. Это приводит к прерывистой анимации и может выглядить неожиданно для игрока. Для исправления этих ошибок и приведения анимации к более плавному, менее неожиданному виду, могут быть использованы методы конвергенции (приближения). Вместо резкого перемещения, мы постепенно корректируем

позицию сущности. Например, мы можем стремиться поставить сущность в нужное место в течение 200 мс, так чтобы через 200 мс она была в нужном месте - для этого не достаточно просто переместить её в эту позицию спустя 200 мс; нам нужно сначала оценить, где она может находиться спустя 200 мс, и постепенно перемещать её в это место.

Первый шаг алгоритма - выбрать место и время исправления позиции сущности (например, новая позиция спустя 200 мс). Сущность затем движется с увеличенной скоростью, пока не достигает нужного места, как показано на рисунке 5а. Каждая такая корректировка представляет собой движение вдоль кривой, как показано на рисунке 5б. Эта прогрессивная корректировка может быть сложна в программировании, поскольку нам нужно иметь дело сразу с текущей позицией сущности, его правильно (но устаревшей) позицией, и с правильной будущей позицией. Более того, позиция персонажа должна обновляться спустя некоторое время до тех пор пока правильная позиция не будет достигнута.

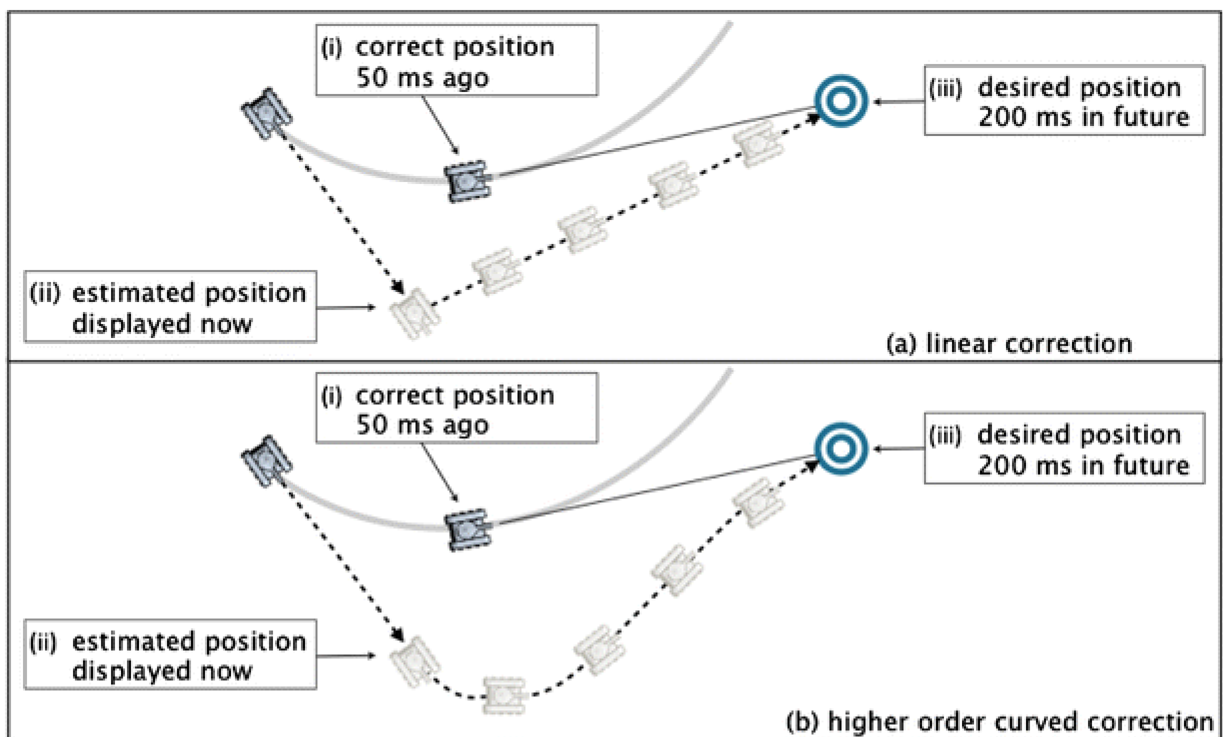


Рис 5

Исправление неточных прогнозируемых состояний Расчета пути может быть скорректировано если А - следовать прямо по пути к будущей спрогнозированной точке; или Б - следовать вдоль кривой.

Плавные исправления позиции могут быть легко реализованы с помощью хронологий. Мы делаем это с помощью переопределения стандарной функции удаленного обновления состояний, показанной на рисунке 4. По умолчанию, удаленные обновления помещают приходящее значение в хронологию по подходящему времени. Этот подход повторяет хронологию у всех игроков, имеющих к ней доступ. Для плавных корректировок мы намеренно хотим чтобы хронологии различались, и когда игрок получает корректировку, локальная хронология изменяется, постепенно приближая состояние к нужному значению.

Рисунок 6 показывает этот подход. Мы используем хронологию для представления позиции персонажа другого игрока. Текущая позиция персонажа экстраполируется, основываясь на данных о известных позициях. Обозначим эту экстраполируемую позицию как currentPos. Далее, как можно увидеть на рисунке 6, клиент получает сообщение, указывающее, что персонаж на самом деле находился в позиции fixupPos в момент времени fixupTime. Экстраполируя эту позицию и время, мы определяем, что персонаж должен на самом деле находиться в позиции correctCurrentPos.

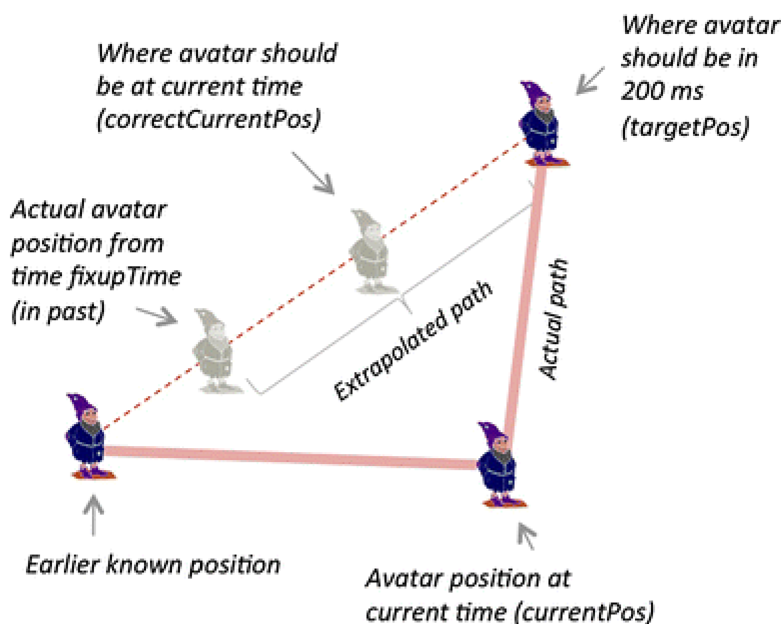


Рис. 6

Хронологическая реализация плавных исправлений.

Простое решение этой ошибки - обновление текущей позиции значением correctCurrentPos, и хронологическая модель обычно делает это сама.

Вместо этого мы приняли решение плавно двигать персонажа в правильную позицию в течение следующих 200 мс. Целевая позиция персонажа (targetPos) определена экстраполяцией на 200 мс в будущее. Персонаж будет быстро двигаться в эту поизицию в течение 200 мс.

Теперь мы опишем более детально как это осуществляется с помощью хронологической модели. Сначала нам нужно сохранить текущую позицию перед применением обновления, то есть позицию во времени 0. (Это необходимо, потому что ввод нового прошедшего значения в хронологию изменит текущую позицию.)

currentPos = avatarPos(0);

Далее мы поместим исправляющую позицию (позицию, содержащуюся в последнем сообщении) в хронологию на текущий момент. Это даст нам оценку целевой позиции, в которую мы хотим попасть спустя 200 мс.

avatarPos(fixupTime) = fixupPos;

targetPos = avatarPos(200);

И наконец, мы поместим обе текущие позиции и целевую позицию в хронологию. (Назначение целевой позиции необходимо так как экстраполируемое значение изменится в ходе обновления текущей позиции)

avatarPos(0) = currentPos;

avatarPos(200) = targetPos;

Корректировка может быть либо по прямому пути к целевой позиции, как показано на рисунке 5а, либо, если используется нелинейная функция интерполяции, корректировка будет вдоль кривой линии как на рисунке 5б.

Этот пример показывает, как легко реализуются Плавные исправления с помощью хронологий. По умолчанию, удаленная функция обновления хронологии просто помещает значение, приходящее от других игроков в локальную копию хронологии, эта функция содержит лишь один шаг:

avatarPos(fixupTime) = fixupPos;

Простым дополнеием этой функции четырьмя строками кода мы получаем следующую функцию обновления:

currentPos = avatarPos(0);

avatarPos(fixupTime) = fixupPos;

targetPos = avatarPos(200);

avatarPos(0) = currentPos;

avatarPos(200) = targetPos;

Вот мы и реализовали плавные корректировки. Кроме того, в зависимости от того, какие функции интерполяции или экстраполяции используются (линейная или высшего порядка), корректировки могут быть как линейными так и по кривой линии.

Ключевой момент, показанный в этом примере - имея возможность изменять хронологию, программист может явно контролировать отклонения (различия) между хронологиями разных игроков. Игрок, который контролирует персонажа, не будет замечать ни ошибок, ни корректировок.

4.2 Пример: Метод отложенного ввода

В то время как Расчет пути - это метод прогнозирования, который помогает избежать несоответствий игрового состояния у игроков, методы отложенного ввода, такие как Локальная задержка и Баскет-синхронизация используются для иных целей. С помощью этих методов обеспечивается сокращение или устранение несоответствий игровых состояний с помощью задержки локальных действий.

Игроки обычно лучше координируют свои действия, если они одновременно видят изменения игрового состояния. Для обеспечения синхронизации действий игроков были разработаны различные алгоритмы манипуляции со временем. Наиболее примечательный среди них - Локальная задержка. Ключевая идея Локальной задержки - откладывание выполнения локальных команд на время, достаточное для распространения сообщений всем удаленным клиентам для того, чтобы выполнить их везде одновременно. Программирование локальной задержки удивительно сложно, требует механизмов для отложенного ввода и для оценки времени передачи сообщения между различными клиентами. А также требует политики для хранения сообщений дольше времени постоянной задержки.

Несмотря на это, Локальная задержка легко реализуется с помощью хронологий. Снова рассмотрим Алису и Боба из нашего предыдущего примера для Расчета пути. Как и раньше, позиция Алисы хранится в

хронологии alicePos. В этом примере клиент Алисы использует Локальную задержку для установки позиции в ответ на команды движения. Предположим, что время постоянной Локальной задержки равно DELAY. Когда игрок Алисы нажимает на кнопку, движение произойдет с отставанием в DELAY мс до того момента, как персонаж начнет выполнять действия, связанные с движением. Боб тоже будет соблюдать похожий принцип движения удаленного персонажа, перемещая его спустя DELAY мс после нажатия на кнопку у Алисы.

Если Алиса передвигается в новую позицию (x, y), действия на клиенте Алисы при движении просты:

alicePos(DELAY) = (x, y);

Таким образом, позиция (x, y) установится в хронологию alicePos на DELAY мс в будущем. Например, если DELAY = 100, тогда позиция Алисы установится в (x, y) спустя 100 мс.

Далее, позиция Алисы используется при рисовании на обоих клиентах как:

DrawAvatar(alicePos(0));

Таким образом, персонаж всегда рисуется в позиции, где он находится на текущий момент времени.

Этот очень простой код имеет ряд интересных эффектов. На клиенте Боба данные о движении Алисы автоматически помещаются в хронологию в момент их прибытия. Если на доставку сообщения ушло менее чем DELAY мс (надеемся на нормальный случай), тогда удаленный клиент Боба поместит новую хронологию alicePos в будущем. Например, если DELAY = 100 и сообщение было доставлено за 60 мс, сообщение поместится в хронологию alicePos на компьютере Боба на момент времени t=40. Это позволяет интерполировать последнюю позицию Алисы (используя ранее записанные позиции). Следовательно, функциональность Локальной задержки поддерживает плавное движение на удаленных клиентах без досадных корректировок.

Напротив, если сообщение доставляется дольше, чем время DELAY, скажем, 130 мс, тогда обновленная позиция поместится в хронологию в прошлом (t =

-30), и текущие позиции будут экстраполируются из этого (и по возможности других) прошлых значений.

Этот простой пример демонстрирует силу хронологической модели. Простое изменение времени при котором в хронологию устанавливается позиция Алисы дает возможность разработчику переключаться с Расчета пути на Локальную задержку. Также, проблемы синхронизации задержек между различными клиентами и ожидание задержек, при которых сообщения доставляются дольше определенной постоянной задержки, и поступление данных на хранение автоматически - всё это не требует дополнительного программного кода.

4.3 МЕТОДЫ СМЕЩЕНИЯ ВРЕМЕНИ (Time-offsetting)

Методы смещения времени визуализируют сущности в различное время на различных клиентах, обычно отображая задерживающуюся (или смещенную по времени) версию удаленного игрока или объекта. Этот подход используется в тех случаях, когда требуется обеспечить точное представление состояния игры во время действий игрока, даже если это представление задерживается.

4.3.1 ПРИМЕР: УДАЛЕННАЯ ЗАДЕРЖКА (remote lag)

Один из методов смещения времени - механизм прицеливания, используемый в играх серии Half-Life. Для принятия решений попал ли стреляющий игрок по целевому игроку или нет, в шутерах обычно используется авторитарный центральный сервер. Простейший вариант реализации этого подхода заключается в том, что имеется одно каноническое игровое состояние, и после того, как был совершен выстрел, сервер определяет позицию целевого игрока и произошло ли попадание. Такой подход может сделать прицеливание сложным, потому что стрелок должен предположить, где персонаж другого игрока будет двигаться и целиться в ту позицию, чтобы в будущем произошло попадание (рисунок 7).

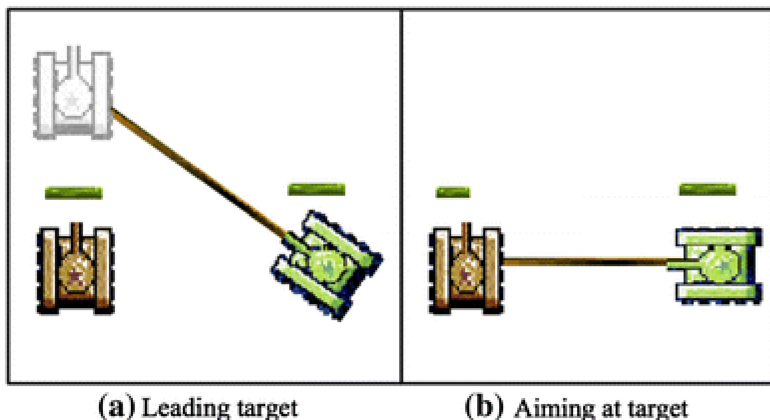


Рис. 7

Стрельба часто требует от игроков "упреждения" - угадывания, где цель находится на самом деле (a). Алгоритм Half-Life позволяет игрокам стрелять по цели в ту позицию, где они её видят (b).

В играх серии Half-Life каждый клиент применяет постоянную задержку для действий других игроков, а сервер принимает решение о попаданиях, основываясь на состоянии клиента-стрелка на момент выстрела (рисунок 7б). Это предполагает, что стрелок может прицеливаться прямо в целевого игрока. Тем не менее, реализация этого механизма может быть сложной, так как серверу требуется иметь возможность отмотать время для определения позиции целевого персонажа на момент выстрела на клиенте стрелка. Поскольку задержка применяется только для действий персонажа удаленного игрока, точки зрения у каждого из клиентов различаются.

Рисунок 8 показывает, как хронологии используются для решения этой проблемы. Сначала мы посмотрим как хронологии используются для упрощения визуализации текущей позиции локального персонажа и смещенной по времени позиции удаленного персонажа. Предположим, имеется две хронологии avatar1 и avatar2. Эти хронологии будут хранить позиции персонажей, направления их прицеливания, и индикатор производимого персонажем выстрела. Без потери общности, мы предполагаем, что avatar1 сейчас стреляет в avatar2. Эти две хронологии распределены между клиентами и центральным сервером (на каждом из них есть своя копия хронологии). Когда игроки выполняют действия ввода, соответствующие хронологии обновляются в момент времени 0. Итак, если персонаж игрока 1 передвинется в позицию (x, y), нацелится в направлении (aim_x, aim_y) и выстрелит, состояние персонажа установится следующим образом:

avatar1(0) = (x, y, aim_x, aim_y, true);

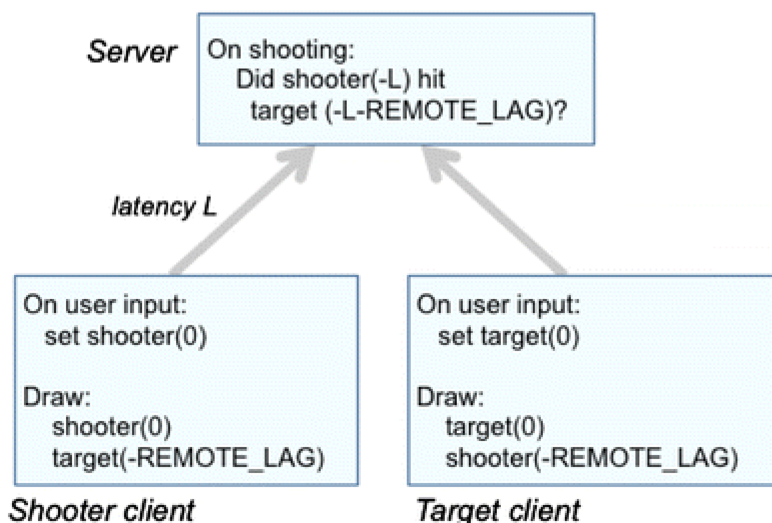


Рис 8.

Хронологический подход к реализации определения попаданий по методу игр Half-Life.

Когда происходит визуализация персонажей, мы хотим изобразить локального персонажа на текущий момент, но изображаем задерживающуюся позицию персонажа удаленного персонажа. Итак, предположим, что задержка перед применением обновлений удаленного персонажа равна REMOTE_LAG, тогда на клиенте игрока 1 персонажи нарисуются так:

DrawAvatar(avatar1(0));

DrawAvatar(avatar2(-REMOTE_LAG));

Теперь мы посмотрим, как хронологии могут быть использованы для обеспечения возможности определить на сервере, целился ли игрок 1 в игрока 2 в момент выстрела. Сначала нам понадобится знать продолжительность времени на доставку сообщения от клиента до сервера. Если мы предположим, что сообщение до сервера доставляется спустя L мс. Тогда сервер должен принять решение о попадании, основываясь на состоянии стрелка (avatar1) во время -L (когда сообщение было отправлено) и на состоянии цели (avatar2) во время -L-REMOTE_LAG (когда стрелок предполагал, где цель была во время -L).

Периодически сервер проверяет хронологию avatar1, чтобы увидеть, был ли сделан выстрел. Время последнего состояния стрелка необходимо и

сохранено в t. Состояние стрелка в это время извлекается и используется для определения момента, в который стрелок произвел выстрел:

t = LastKnownTime(avatar1);

shooter = avatar1(t);

if (shooter.isShooting) ...

Если игрок выстрелил, сервер определяет видимое стрелком положение цели в момент выстрела. Оно получается вычитанием удаленной задержки из времени выстрела. Положение цели в это время будет следующим:

target = avatar2(t - REMOTE_LAG);

Наконец, функция TargetHit использует позиции и направления персонажей для определения, произошло ли попадание в цель. Если попадание произошло, количество жизней у цели уменьшается.

```
if (TargetHit(shooter.Position, shooter.Heading, target.Position))
{
    targetHealth(0) = targetHealth(0) - 1;
}
```

Итак, сервер должен иметь доступ к точке зрения стрелка на момент времени, когда был произведен выстрел, для определения попадания в цель. Для того, чтобы это сделать, сервер должен воссоздать состояние клиента-стрелка на момент выстрела. Это требует от сервера доступа к состоянию стрелка в момент выстрела и доступа к состоянию цели в точке времени, предшествующей выстрелу, в соответствии с тем, что видел стрелок. Хронологическая модель обеспечивает легкий доступ к этим прошлым состояниям и делает процесс простым. Прошлые состояния могут извлекаться из хронологии простым указанием времени из которого нужны данные. Таким образом, для сервера Расчет попаданий требует только знания о времени, когда был произведен выстрел и времени задержки соединения сервера с клиентом.

4.3.2 Пример: Локальные фильтры восприятия (local perception filters)

Локальные фильтры восприятия - другой пример алгоритма лагокомпенсации, основанного на смещении времени. Ключевая идея заключается в постоянной регулировке задержки применительно к неконтролируемым игроками сущностям в зависимости от их позиции относительно персонажа игрока. Несмотря на то, что этот подход очень многообещающий, мы не знаем ни одной игры, использующей Локальные фильтры восприятия. Мы догадываемся, что это может быть из-за сложности реализации его при использовании стандартных средств программирования, делающих его сложным для оценки разработчиками применимости этого алгоритма в их игре. Как мы покажем далее, хронологическая модель делает реализацию Локальных фильтров восприятия податливой, открывая возможности к распределенной игровой физике.

Сначала мы опишем алгоритм Локальных фильтров восприятия и потом покажем как он может быть реализован с использованием хронологий. Мы начнем с рассмотрения одиночного объекта, движение которого определяется физическим движком, например, в простой футбольной игре как показано на рисунке 9. Посредством этого примера мы осветим несколько недостатков других часто используемых методов лагокомпенсации. В футбольной игре два игрока пинают мяч в 2D мире. Игроки свободно передвигают своих персонажей в мире и симуляция физики используется для определения позиции мяча. В идеале, оба игрока должны видеть мяч в одной и той же позиции и мяч должен реагировать сразу же во время удара. Однако, из-за сетевых задержек это невомзожно. Метод отложенного ввода, такой как Локальная задержка должен обеспечить желаемое согласованное представление, но игрок должен испытывать меньшую задержку чтобы иметь возможность ударить по мячу во время его движения. Методы предсказания обеспечивают быстрый отклик, но экстраполируемые значения часто бывают неточными, особенно, если персонаж меняет направление или пинает мяч. Когда присутствует имитация физики, всякая неточность становится очень заметной.

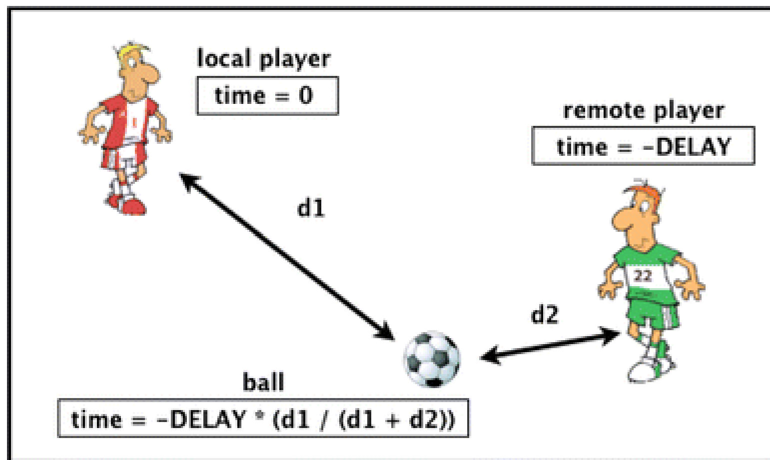


Рис. 9

Когда используются Локальные фильтры восприятия, время для извлечения позиции мяча изменяется относительно локального и удаленного игроков.

Удаленная задержка (как описано выше) применяется для удаленного игрока и визуализации локального персонажа в реальном времени. Это обеспечивает немедленный отклик для локального игрока. Поскольку позиция удаленного персонажа задерживается, обновления позиций обрабатываются как будущие значения, позволяя позиции интерполироваться на локальном клиенте, поддерживая плавную анимацию. Поскольку игрок перемещается между персонажами двух игроков, когда удаленный персонаж пинает мяч, движение мяча должно быть задержано на некоторый промежуток времени, как и удаленного персонажа. Иными словами, взаимодействия между игроками и мячом будет нереалистичным. Однако, когда мяч находится близко к персонажу локального игрока, он должен двигаться в реальном времени, иначе взаимодействия снова будут нереалистичными. Локальные фильтры восприятия балансируют это, регулируя время задержки мяча в зависимости от позиции относительно двух персонажей. Например, если удаленный персонаж отстает на 100 мс, мяч тоже отстает на 100 мс когда он рядом с персонажем удаленного игрока, на 50 мс - если он на полпути между двумя персонажами и на 0 мс когда он близко к персонажу локального игрока. Это обеспечивает локального игрока возможностью быстро реагировать когда он бьет по мячу. Также это делает возможным реалистичное представление взаимодействия с мячом удаленного персонажа.

Для реализации этого примера разработчку необходимо постоянно регулировать задержку, связанную с мячом, таким образом, чтобы она зависела от позиций персонажей. Как дополнительная сложность -

симуляция физики мяча должна осуществляться на клиенте, чей персонаж ближе к мячу. Задержка в визуализации мяча на других клиентах обеспечивает достаточное время для передачи данных о позиции мяча по сети. В итоге, для реализации этой простой игры, использующей Локальные фильтры восприятия, программист должен поддерживать различных временные рамки для обоих игроков, регулировать эти временные рамки динамически и реализовать распределенную симуляцию физики с динамическим изменением клиента, на котором происходит рассчет физики.

Хронологии делают эту реализацию легко поддающейся, потому что они дают программисту легкий доступ к состояниям в любой момент времени. Конкретнее, мы используем хронологии для:

- передачи права рассчитывать позицию мяча, используя симуляцию физики;

- определять время задержки для мяча;

- определять время задержки для персонажа другого игрока.

В нашем примере мы предполагаем клиент-серверную архитектуру с двумя клиентами, каждый из клиентов запускает симуляцию физики для определения позиции мяча. Сервер использует позиции мяча и персонажей для решения, какой из игроков вычисляет позицию мяча.

Для демонстрации того как хронологии могут быть использованы для реализации Локальных фильтров восприятия, нам необходимо иметь три хронологии: player1Pos, player2Pos и ballPos, представляющие собой позиции трех игровых сущностей. Мы также должны иметь одну дополнительную хронологию ballControl, которая указывает какой игрок контролирует расчет физики мяча и обновление хронологии ballPos. Хронология ballControl содержит дискретные значения и, таким образом, использует шаговые интерполяцию и экстраполяцию.

Сервер получает обновления для позиций игроков и мяча из каждого клиента и затем, опираясь на дистанцию между каждым из игроков до мяча, определяет какой игрок должен обновлять позицию мяча. Сервер затем устанавливает эту информацию в хронологию ballControl.

ballControl(0) = Player1;

Следующие шаги происходят на компьютерах каждого из игроков. Здесь мы описываем шаги, при условии, что игрок 1 - локальный игрок. Тогда на

компьютере 1 игрока происходит проверка значения в хронологии ballControl. Если игрок 1 - игрок, который должен контролировать мяч, его клиент рассчитывает новую позицию мяча и устанавливает значение в ballPos. Если игрок 2 - игрок, который должен контролировать мяч, клиент игрока 1 ничего не делает и это даст возможность получить значения из хронологии, которые поместит туда клиент игрока 2. Итак, предположим, что getUpdatedBallPos - это метод, обновляющий позицию мяча и затем возвращающий его обновленную позицию. Тогда код добавления новых значений в хронологию будет таким:

```
if (ballControl(0) == myPlayerNumber) {
    ballPos(0) = getUpdatedBallPos();
}
```

Далее, для вывода мяча в правильной позиции, игроки должны определить задержку относительно мяча.

Предположим далее, что игрок 1 - локальный игрок, тогда персонаж игрока 2 нарисуется как DELAY мс в прошлом. Его позиция будет следующей:

```
player2Pos(-DELAY);
```

и персонаж локального игрока нарисуется в позиции:

```
player1Pos(0);
```

Для оценки позиции мяча мы используем позицию, в которой он рисовался в последний раз как первое приближение. Мы будем полагать, что это значение было сохранено в переменной под названием prevBall. Тогда, предположим что задержка применения для мяча - линейная зависимость от дистанции между ним и двумя персонажами игроков, мы можем рассчитать ballDelay как следующей выражение:

```
d1 = length(player1Pos(0) - prevBall);

d2 = length(player2Pos(-DELAY) - prevBall);

ballDelay = DELAY * d1/(d1+d2);
```

Тогда позиция мяча будет следующей:

```
ballPos(-ballDelay);
```

Этот пример показывает как хронологии дают способность для доступа к позиции сущности в любой момент времени, позволяя нам легко изменить временные рамки мяча для каждого момента визуализации. Он также показывает как дать возможность нескольким клиентам координировать обновления для общих данных. Дает легкость, с которой локальные фильтры восприятия могут быть реализованы с хронологиями, что позволяет определить применимость данного подхода во время разработки игры.

4.4 Применение хронологий: выводы

В этой секции мы доказали, что алгоритмы лагокомпенсации сводятся к трем методам - предсказанию данных, откладыванию действий и смещению времени. Мы показали, что хронологии могут быть использованы для быстрого представления алгоритмов из всех трех методов. Мы также показали, что сложные алгоритмы могут быть выражены в очень маленьком коде, что делает их податливыми для разработчика и для экспериментов со сложными и инновационными схемами лагокомпенсации.

5. РЕАЛИЗАЦИЯ ХРОНОЛОГИЙ

Предыдущий раздел показал, как хронологии работают со временем для обеспечения необходимой инфраструктуры для реализации множества методов лагокомпенсации, используемых в многопользовательских играх. Теперь мы поговорим о том, как хронологии реализуются внутри нашего Janus toolkit. Janus предоставляет низкоуровневую реализацию хронологий с несложным API. Этот инструмент называется именем Януса - персонажа Римской мифологии, повелителя путей, начал и завершений. Янус также имел способность смотреть одновременно в прошлое и будущее, также как пользователи Janus toolkit могут иметь доступ к предыдущим и будущим версиям игровых состояний.

Janus toolkit написан на C# и подходит для любого .NET совместимого языка. Он построен на популярной сетевой библиотеке Lidgren (http://code.google.com/p/lidgren-network-gen3), которая обеспечивает надежную передачу сообщений по протоколу UDP.

5.1 Объектная модель

Внутри Janus хронологии реализованы как объекты класса Timeline. Каждая хронология имеет общий тип (тип хронологического состояния) и методы реализации интерполяции, экстраполяции и обработчик удаленного

обновления. Обычные значения этих методов могут быть перезаписаны для создания произвольных типов хронологий. Хронологические объекты также обеспечивают Get и Set методы для получения/изменения хронологических состояний.

Каждый объект хронологии имеет строковый идентификатор. Если два клиента создают хронологии с одинаковыми идентификаторами, тогда хронолгии автоматически синхронизируются. Как показано на рисунке 4, всякий раз, когда новое значение добавляется в хронологию, обновление посылается по сети к удаленным клиентам. Когда удаленное обновление получено, оно применяется к хронологии через функцию удаленного обновления. По умолчанию эта функция просто помещает новое состояние в хронологию для правильного времени и удаляет любое старое значение в хронологии. Как мы видели, перезапись этой функции может дать возможность легко запрограммировать интересные поведения, такие как плавные исправления.

Известные состояния хронологии (те, которые были помещены в хронологию с помощью метода Set) - просто организованы как двунаправленный список, где каждое состояние помечено его временем. Состояние общих данных может быть просто как одиночное целое число или может быть любым произвольным сложным объектом, хранящим множество свойств. Мы создали разнообразные стандартные хронологии, такие как целочисленная, нецелая, 2D и 3D векторы позиции и более сложные объекты, соединяющие позицию, направление и скорость. Пользователи инструмента могут просто использовать эти хронологические объекты или создавать новые типы хронологических объектов.

Хотя временные значения хранятся в связанном списке как записи о реальном времени (измеряемые в мс от старта программы), программист всегда имеет доступ к состояниям, используя относительное время, где 0 означает текущее время, +10 означает на 10 мс в будущем, и -10 - на 10 мс в прошлом.

5.2 Интерполяция и экстраполяция

Метод Get в хронологиях используется для извлечения состояния для указанного времени (прошедшего, текущего или будущего). Метод Get использует функции интерполяции и экстраполяции хронологических значений как необходимые для предоставления значения для времи, в

которое значение явно не известно. По умолчанию реализация линейной и ступенчатой интерполяции/экстраполяции предоставлена, но разработчики могут создавать дополнительные функции для предоставления особого поведения. Например, с Расчетом пути функция экстраполяции может быть любой первого порядка (основанной на позиции и скорости) или второго порядка (основанной на позиции, скорости и изменениях скорости и направления). Функция может также быть основана на самом позднем обновлении, или может быть основана на двух или более предыдущих состояниях. Наша реализация хронологий поддерживает использование различных функций экстраполяции (и интерполяции) и все настройки доступны просто выбором другой экстраполяционной функции. Какая форма функции экстраполяции наиболее подходит - зависит от типа игры и от типа движения. С хронологиями выбор функции может происходить прямо во время работы программы, что облегчает методы адаптации такие как использование истории позиций, где функция экстраполяции меняется, исходя из движения сущности.

5.3 Распределение и сетевая часть

С точки зрения разработчика, Janus имеет архитектуру одноранговой сети (сеть основана на равноправии участников). Таким образом обновления автоматически транслируются всем участникам имеющим общие хронологии и все данные полностью повторяются. Если требуется сервер (как в примере со стрельбой на рисунке 8), один из участников может быть определен в роли сервера.

В текущей реализации Janus, мы разработали централизованный роутер сообщений для обеспечения одноранговой коммуникации. Роутер основан на архитектуре распределенной публикации и подписках. Как указано ранее, строковый идентификатор привязан к каждому экземпляру объектов хронологии. Когда клиент создает объект хронологии с данным идентификатором, идентификатор посылается в сообщении роутера и клиент автоматически подписывается на обновления этого объекта. Когда клиент помещает новое значение в хронологию, используя метод Set, это значение и время, связанное с ним, передаются в сообщении роутера, который передает данные всем другим клиентам, кто подписан на объект этой хронологии.

Стандартно Janus toolkit делает немногое для того, чтобы уменьшить число и размер сообщений, передаваемых по сети между клиентами. Однако,

некоторые возможности доступны для того, чтобы значительно снизить требования к пропускной способности в программах, использующих этот инструмент. Во первых, программист может установить минимальный временной интервал между обновлениями. Таким образом, не все изменения локальной хронологиии будут передаваться по сети. Например, если локальный клиент обновляет позиции каждые 20 мс, и минимальный интервал между отправкой обновлений будет установлен в 60 мс, то только одна треть обновлений будет передаваться. Во-вторых, каждый клиент продолжает отслеживать какие значения на самом деле были отправлены по сети. Затем, перед отправкой обновления, клиент выполняет проверку, чтобы определить, смогут ли удаленные клиенты точно спрогнозировать новое обновленное состояние. Новое состояние будет отправлено, только если удаленный клиент не может спрогнозировать новое состояние в пределах установленного порога ошибки. За счет установки величины порога ошибки, программист может контролировать долю отправляемых сообщений. Автоматически адаптирующиеся схемы Расчета пути также могут быть реализованы с помощью регулировки порога ошибки в зависимости от игровых ситуаций или факторов, таких как нагруженность сети или пропускная способность.

Наконец, программист может оптимизировать размер и формат сообщений. По умолчанию, Janus использует объектную сериализацию и конвертирует объекты в массив байт для передачи по сети. Это дает преимущество для упрощения создания новых классов хронологий без необходимости волноваться о том, как данные будут переданы. Тем не менее, встроенная объектная сериализация может генерировать неприемлемо большие сообщения. Программисты могут перезаписать стандартный метод сериализации и таким образом подобрать оптимальный формат для передачи сообщений.

Для обеспечения синхронизации Janus использует глобальные часы. Определение и поддержка глобальных часов на клиентах может быть пугающей задачей из-за смещения часов, сетевых задержек и рывков. Мы реализовали наши глобальные часы с помощью алгоритма Беркли. В нашей реализации, роутер сообщений периодически передает текущее время всем клиентам. Роутер сообщений анализирует полученные от клиента сообщения, содержащие время, отбрасывая любые посторонние данные, и затем передает обновленное глобальное время обратно клиенту.

5.4 Альтернативные архитектуры

Хронологическая модель не навязывает какую-либо специфическую архитектуру игре. В Janus toolkit мы выбрали для реализации обмена сообщениями одноранговую архитектуру. Решение использовать центральный роутер сообщений было принято чисто из-за простоты реализации и может быть заменено настоящим методом однорангового обмена сообщениями. Эта реализация легко поддерживает разнообразные архитектуры. Как показано на рисунке 10, архитектура взаимодействия сообщениями определяется топологией подписки клиентов на объекты хронологий. Рисунок 10а показывает одноранговый принцип подписки каждого клиента (C1 и C2) на обновление всех хронологий (TL1 и TL2). В клиент-серверной модели, показанной на рисунке 10b, каждый клиент передает данные о хронологиях только серверу. Это значит, что клиент C1 обновляет хронологию TL1 и роутер сообщений распространяет эти обновления только серверу S1. Подобно этому, клиент C2 обновляет хронологию TL2 и роутер сообщений передает обновления только серверу S1. Хронологии TL3 и TL4 одновременно обновляются сервером и обновления этих хронологий пересылаются обоим клиентам. В гибридной модели (рисунок 10с), хронологии TL1 и TL2 распределены только между одним клиентом и сервером, в то время как хронологии TL3 и TL4 распределены между двумя клиентами и хронология TL5 распределена между всеми.

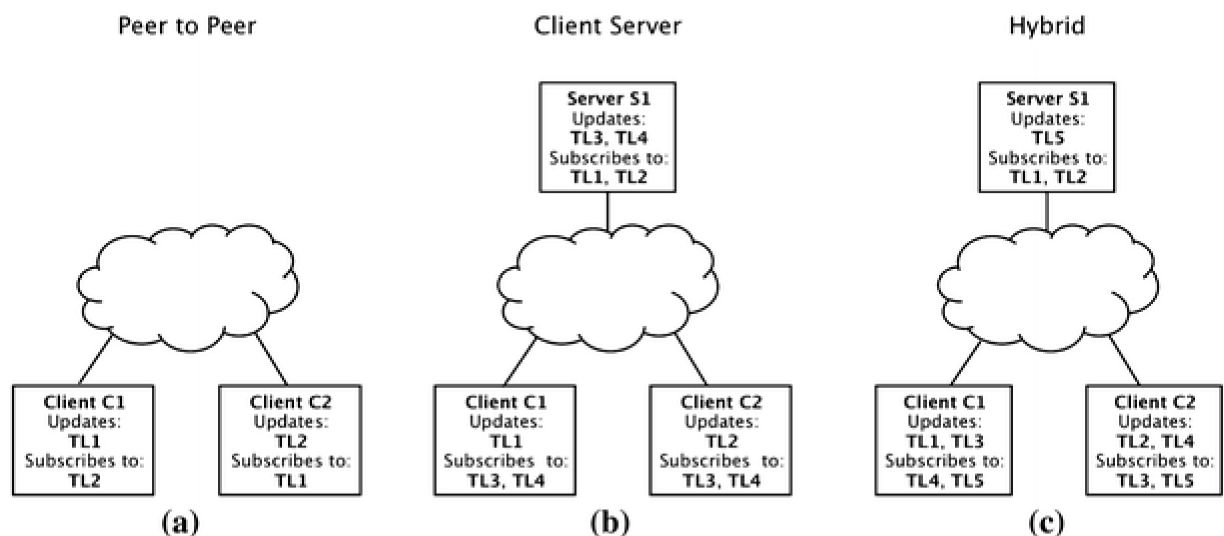


Рис. 10

Janus toolkit позволяет использовать разнообразные хронологии, включая одноранговую, клиент-серверную и гибридную модель, в зависимости от выбранного варианта подписок на хронологии.

6 ФОНОВАЯ И СМЕЖНАЯ РАБОТА

Мы показали, что сделав вермя - одной из составных частей модели программирования, хронологии упрощают реализацию методов лагокомпенсации, используемых в многопользовательских играх. Чтобы принять эти сведения в работу, мы теперь рассмотрим предоставление поддержки существующих сетевых библиотек для такой модели программирования, и затем рассмотрим другие программные среды, связанные со временем.

6.1 Сетевая поддержка для многопользовательских игр

Существующие инструменты для организации сети в играх имеют ограниченную поддержку для манипуляций со временем, способствуя усложнению реализации алгоритмов лагокомпенсации. Zoid-Com включает специальный репликатор, который реализует прогнозирование на стороне клиента, Расчет пути, экстраполяцию, интерполяцию, плавные исправления движения и локальную перезапись. Как мы видели, интерполяция и экстраполяция - наиболее важные составляющие алгоритмов лагокомпенсации, но их недостаточно для полного решения проблем, описанных в секции 2, особенно это касается проблем точек зрения и множества моментов в одном.

OpenTNL также имеет механизмы интерполяции и экстраполяции позиций объектов. Net-Z поддерживает две модели общих данных - распространение атрибутов, которые используют данные экстраполяции для уменьшения нагрузки на сеть, и пошаговую синхронизацию. Другие сетевые библиотеки, такие как ClanLib, Nevrax/NEL, OpenSkies, RakNet и ReplicaNet поддерживают базовые сетевые сервисы и включают копирование объектов, NAT-прорыв (сетевую перфорацию), надежную доставку сообщений и методы сокращения нагрузки на сеть. Тем не менее, все они поддерживают малую часть или вовсе не поддерживают интерполяцию и экстраполяцию для общих данных.

Так как сетевые библиотеки поддерживают ограниченный набор возможностей для программирования времени, мы посмотрим другие

программные среды, которые имеют работу со временем, встроенную в модель программирования.

6.2 Другие среды программирования для работы со временем

Рассмотрим несколько языков программирования, включающих в себя время. Языки программирования потоков данных (такие как Lucid) представляют переменные и выражения в виде бесконечных серий объектных данных как обычные простые значения. В языках потоков данных переменные относительно смещаются из одного состояния в следующее; тем не менее, там нет механизма для непосредственного доступа к состояниям в прошлом и будущем. Императив-содержащие языки расширяют языки потоков данных для представления временных ограничений в пользовательском интерфейсе, но снова не дают возможности работать с прошлыми и будущими состояниями.

Доступны также некоторые коммерческие языки, имеющие отношение ко времени. Quicktime обеспечивает обширную поддержку для медиа-данных, основанных на времени. Такие инструменты, как Adobe Flash, Core Animation и Windows Presentation Foundation обеспечивают явный доступ ко времени как инструмент создания анимации. Они позволяют устанавливать атрибуты анимации (такие как позиция или цвет) в двух точках времени. Это дает возможность получить интерполируемые значения в любой точке между стартовой и финишной. Тем не менее, программист ограничен в доступе к данным из разных точек анимации одновременно, а также не существует понятия «распределения анимаций» между участниками, соединенными по сети.

Пространственно-временные базы данных охватывают пространственные и временные аспекты данных и имеют дело с позицией и/или геометрией объектов, меняющихся во времени. Пространственно-временные базы данных поддерживают запросы о времени, временных свойствах и временных отношениях, позволяя получить доступ к данным в любой момент времени. Мы смогли извлечь кое-что из этих принципов. Однако принцип внедрения базы данных не практичен для игр реального времени, где необходимо локальное копирование и немедленный доступ к данным.

Вычисление изменений в общих данных в зависимости от времени - фундаментальный принцип в распределенных симуляциях. Может использоваться как пошаговая модель, так и событийная. Такие стандарты, как IEEE стандарт Высокоуровневой Архитектуры обеспечивают протокол объектной модели, совместимость которой включает в себя измерение (dimension) времени и спецификации для экстраполяции (Расчет пути).

Пожалуй, самая близкая к хронологиям программная абстракция предложена в системах "Историки процесса", таких как OSISoft's PI System и AspenTech's InfoPlus.21. Они используют в процессе управления производством хранение данных и событий во времени. API этих систем реализует множество принципов, необходимых для программирования с учетом состояний и времени, включая способность устанавливать и получать значения в произвольном времени и автоматически интерполировать значения между временными интервалами. Данные могут быть получены, используя как абсолютное, так и относительное время. "Историки процесса" направлены на совсем другую область, область управления процессами, и не рассчитаны для использования в распределенных системах.

Эти инструменты и языки программирования представляют различные принципы для работы с данными, меняющимися во времени. Наша работа расширяет временные компоненты, найденные в этих областях, за счет применения их к распределенным данным в сети. Конкретнее, наша хронологическая модель объединяет способность Flash и WPF для индексации переменных во времени и способность пространственно-временных баз данных и "Историков процессов" для установки и выбора данных в указанное время в прошлом и будущем, и применяет эти принципы к распределенным данным в форме, пригодной для использования в сетевых играх.

В меру наших знаний, наша хронологическая модель и её реализация в Janus toolkit уникальна. Это первая модель программирования для распределенных данных, которая включает в себя время и состояния. Делая время явным измерением для распределенных данных, хронологическая модель значительно упрощает реализацию широкой области алгоритмов лагокомпенскации, используемых в многопользовательских играх.

7. ОБСУЖДЕНИЕ

Теперь мы опишем наш опыт использования хронологий, их сильные стороны и ограничения этой модели программирования.

7.1 Опыт разработчиков

Несмотря на статус научного прототипа, Janus toolkit был использован авторами и другими разработчиками для экспериментов с разными алгоритмами лагокомпенсации и для создания некоторых многопользовательских игр, основанных на библиотеке игровых разработок Microsoft XNA. Такие игры как Balloon Burst, Truck Pull и Pedal Race exergames, трехмерный шутер Eliminate, гоночная игра Speed Demons и симулятор Growl Patrol, а также игра симулятор создания мира Liberi. Инструмент также был использован для реализации военного симулятора OrMiS.

Эти игры разрабатывались десятью разработчиками, никто из них лично не был автором Janus. Все они были студентами, начиная от бакалавра и заканчивая доктором физических наук, и многие из них имели только небольшое представление о программировании распределенных систем. Несмотря на это, все они отметили простоту реализации работы с сетью при использовании Janus toolkit. Для Balloon Burst, Truck Pull, Pedal Race, Growl Patrol и OrMiS время на реализацию работы с сетью заняло несколько часов. Liberi имела сложные требования к производительности, связанные с её огромным, полностью изменяемым миром. Для Liberi Janus использовался в рамках реализации распределенных физических алгоритмов и управления. Несмотря на это, решение вопроса реализации многопользовательской игры заняло всего несколько недель.

Разработчики, использовавшие Janus toolkit, прежде всего, были заинтересованы в создании простых сетевых игр с возможностью игры по локальной сети. В основном они полагались на Прогнозирование и Локальную задержку, или комбинацию этих двух технологий. За исключением одного студента четвертого курса бакалавра, который реализовал три игры с целью оценки эффекта различных методов лагокомпенсации на пользовательском опыте и производительности. В каждой из этих трех игр была возможность переключаться между локальной задержкой, удаленной задержкой и прогнозированием прямо во время игрового процесса. Также в этих играх были реализованы Плавные исправления.

Хотя Плавные исправления относительно легко реализуются с помощью Janus toolkit, многие разработчики решили не использовать этот метод в своих играх. Одна из основных причин этому - игра только по локальной сети, где проблем с исправлением резких изменений не возникало. Включение Плавных исправлений как возможности для перезаписи стандартных функций обновления состояний в классах хронологий позволяет повысить их применимость с использованием Janus для многих начинающих разработчиков.

Наш опыт показывает, что хронологии могут упростить разработку основной работы по сети в многопользовательских играх. Как мы видели, хронологии также делают податливой реализацию сложных алгоритмов. Например, мы экспериментировали с алгоритмом локальных фильтров восприятия и плавными исправлениями для обеспечения инновационного решения проблем распределенных вычислений физики. Далее, мы смогли добиться реалистической симуляции более чем у четырех игроков, взаимодействующих с десятками объектов при сетевом соединении с более чем 100мс задержкой передачи данных. Мы продолжаем работу над увеличением количества игроков и объектов.

Исходя из этого опыта, мы ожидаем большую выгоду от хронологий в том, что они дадут разработчикам быстрый доступ к различным выгодным алгоритмам, а также к созданию новых алгоритмов и комбинированию их в инновационные решения. Если в теории такая работа возможна и с традиционными методами, она не всегда практична, учитывая сжатые сроки разработки коммерческих игр.

7.2 Опыт игроков

Продвинутые методы лагокомпенсации видятся многообещающими для улучшения пользовательского удобства в игровом процессе. Например, в серии игр Half-Life, метод Удаленной задержки повысил удобство обоих игроков и повысил их производительность за счет предоставления возможности игрокам прицеливаться прямо в цель. Алгоритм баскет-синхронизации был ключевым для реализации игры Age of Empires. Однако не всегда можно уверенно сказать, какие методы подойдут лучшим образом для каждой конкретной игровой ситуации. Как показано в работе Стукеля и Гутвина, Пантеля и Вульфа, Зао и других, а также нами самими, эффективность алгоритмов можно оценить в результате тестирования. Для

того чтобы это сделать, разработчик должен выбрать алгоритм, реализовать его и затем оценить его воздействие на производительность игрока и удобство использования во множестве игровых ситуаций. Если метод слишком сложный или объемный в реализации, это создает значительное препятствие для его оценки. Например, локальные фильтры восприятия сначала появились в литературе в 1998 году, но (до разработок, описанных в этой статье) никогда не были реализованы ни в какой многопользовательской игре. Хронологии делают реализацию этих методов гибкой и могут дать разработчикам возможность экспериментировать с различными методами и подстраивать методы под определенные ситуации в игре.

7.3 Сильные стороны и ограничения

Сильная сторона хронологической модели состоит в её явном обращении со временем. Методы автоматической интерполяции и экстраполяции позволяют программистам легко получить доступ к общим игровым состояниям из любого момента в прошлом и будущем. Этот метод является мощным в плане манипуляций с общими данными, хотя не обходится без ограничений. В нашем примере реализации локальных фильтров восприятия, мы показали, как два клиента могут координировать обновление общей хронологии, представляющий позицию мяча. Главное, на что следует обратить внимание, - текущая реализация хронологии не поддерживает обновление одной хронологии множеством клиентов, поскольку обновление от одного клиента по умолчанию может перезаписать обновления, сделанные другим клиентом. Перезапись стандартной функции удаленного обновления может решить эту проблему синхронизации. Однако на данный момент реализация этой возможности ложится на разработчика, использующего Janus toolkit. В будущих версиях инструмента мы предложим ряд настроек для функций удаленного обновления, которые позволили бы обеспечить такие возможности синхронизации, как временной сдвиг (time warp), протокол оптимистической синхронизации (optimistic synchronization protocols), объединение конфликтов (conflict merging) или операционные трансформации (operational transform).

Стандартно хронологии передают по сети все данные для общих объектов при каждом обновлении. Это делает их неподходящими для больших структур данных. Мы начали проводить исследования, как сделать хронологии более эффективными с помощью передачи только обновлений общих состояний вместо передачи целого объекта. Использование

измененных методов сериализации позволяет передавать только часть данных, которая была изменена. В дальнейшем мы планируем провести работу для применения этого решения ко всем хронологическим объектам.

Мы показали, как хронологическая модель облегчает доступ к общим данным в любой момент времени. Однако это не подходит для логических типов данных, которые описывают свойства-команды "стреляет" или "присел". Не существует метода интерполяции или экстраполяции этих типов действий и каждое такое свойство должно быть доступно по отдельности. Мы экспериментировали с различными вариантами включения подобных свойств-команд в нашу хронологическую модель. В Janus toolkit доступны некоторые настройки, позволяющие программисту получить список команд, вызванных в течение некоторого промежутка времени или для использования событийной модели для команд, возможность отложенных событий основана на временной метке, привязанной к команде.

Наша реализация глобальных часов была успешно применена для синхронизации клиентов на различных компьютерах. Мы нашли удачным решение синхронизации глобальных часов по локальной сети спустя каждые несколько миллисекунд. Тестирования в широкополосных (глобальных) сетях показало, что синхронизация часов каждые несколько десятков миллисекунд достижима. Многие сложные алгоритмы могут быть необходимы, в том числе и в условиях больших сетевых задержек и скачков. Кроме того мы не организовали защитные меры, гарантирующие, что изменение времени в часах будет происходить постепенно и что часы не смогут пойти в обратную сторону. Мы не видим необходимости в таких защитных мерах на данный момент, однако возможно этот вопрос потребуется рассмотреть в дальнейшем.

Количество памяти, используемое для нашей реализации представляет собой область будущей оптимизации. На данный момент все значения, которые были установлены, сохраняются в хронологии, что в свою очередь требует большого количества памяти. Janus на данный момент отсекает историю до требований ограничения хранилища. Мы планируем адаптировать алгоритмы для уплотнения истории, разрабатывая решение проблемы запаздывания группы (groupware latecomer problem) и механизмы для сжатия сообщений.

Хронологии предоставляют новую модель программирования. Для программистов, знакомых с технологиями передачи сообщений, переход к

пониманию модели общих состояний, индексируемых по времени, может оказаться значительным, возможно аналогичным переходу от процедурного к объектно-ориентированному программированию. Мы заметили, что разработчики, которые погружаются в модель без внимательного изучения её документации и примеров, допускают ошибки в попытке представить её как систему передачи сообщений. Как и со всеми инновационными моделями программирования, разработчикам нужно привыкнуть к особому типу мышления, соответствующему этой модели.

v. 1.22 (26.12.19)

Perfect Light team

2019