МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования

**«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ Н.Г.ЧЕРНЫШЕВСКОГО»**

Кафедра Математического и компьютерного моделирования

Разработка системы автоматического тестирования игры

АВТОРЕФЕРАТ БАКАЛАВРСКОЙ РАБОТЫ

студентки 4 курса 441 группы

направление 09.03.03 — Прикладная информатика

механико-математического факультета

Азизова Михаила Витальевича

Научный руководитель
к.ф.-м.н., доцент О.М. Ромакина

Зав. кафедрой
зав. каф., д.ф.-м.н., доцент Ю.А. Блинков

Саратов 2019

Введение. В современном мире игровая индустрия заняла стабильную нишу в сфере мировой экономики. Если на начальном этапе развития этой сферы стоимость разработки игры была не очень высокой, то сегодня для создания игр требуется большой штат разработчиков, тестировщиков, художников и т.д. Стабильный спрос на компьютерные игры стимулирует инвесторов делать вложения в развитие данной сферы. Для уменьшения издержек производства игры и соответственно увеличения прибыли, некоторые процессы необходимо автоматизировать. В работе рассматривается создание механизма тестирования уровней игры, для ускорения процесса разработки.

Задачи:

- Анализ способов решения задачи.
- Реализация основной логики алгоритма.
- Создание интерфейса для взаимодействия с разработчиками.

Основное содержание работы. На рынке существует разные жанры видео игр и для каждого отдельного жанра существует свой процесс разработки. В работе рассматриваются процессы игры в жанре TimeManagment. Цель игры заключается в самом оптимальном прохождении уровня. Игрок назначает команды персонажам, по средствам клика на объект, после чего персонажи расчищают путь до следующего объекта. Чтобы пройти уровень, нужно выполнить все задачи. Зачастую задачи представляют собой объекты которые нужно собрать. Также у игрока есть ресурсы, он их получает или тратит в процессе игры. Например, за расчистку груды камней, игрок получает некоторое количество ресурса - камень.

В играх этого жанра процесс разработки построен вокруг создания сложных уровней и проработки баланса в них. Ведь без хороших уровней не получится увлекательной игры. Этой частью работы занимаются левел-дизайнеры. В основном время затраченное на создание одного уровня зависит от проектирования движка игры. Если в среде в которой работают дизайнер, существуют различные инструменты упрощающие определенную работу или автоматизирующие часть работы полностью, уровни будут готовы в разы быстрее. Одним из инструментов является тестировщик уровня. Инструмент позволяет левел-дизайнеру получать результаты прохождения уровня, без

надобности тратить время на прохождение. Также решение будет отображать ошибки уровня. Это очень полезно при автоматической сборке проекта. Проблему создания данного инструмента можно описать в следующих пунктах:

- На уровне существуют цели, по завершению которых, уровень будет считаться пройденным.
- Нет прямого сценария прохождения уровня, игрок может пройти уровень разными комбинациями.
- Персонажи могут работать параллельно.
- Объекты разделены по типу персонажа который их может подобрать.
- Объекты могут требовать или выдавать разные типы ресурсов.

Учитывая все выше перечисленное, необходимо создать решение, которое способно рассчитывать результат игры, затрачивая минимальное количество времени и технических ресурсов.

Задачу можно представить в виде графа. Где структуры это вершины, ресурсы - это некоторые параметры при которых можно обработать вершину. Время сбора плюс время затраченное на путь - это цена вершины. Имея граф игры решение задачи сводится к обходу графа с использованием своего алгоритма. Для того, чтобы оптимально обходить граф объектов, потребуется алгоритм поиска путей A*. В стандартом виде алгоритм можно разделить на несколько шагов:

- поиск соседей и добавление их в список на обработку.
- определение до какой вершины выгодней дойти в текущей итерации.
- обработанные вершины необходимо поместить в список обработанных вершин, чтобы не повторять обработку несколько раз.

Граф представляет собой вершины и связи между ними, в данном алгоритме потребуется тип данных, который будет содержать информацию о вершине:

- Position - позиция точки.
- Neighbours - ссылки на соседние точки, массив ссылок на объекты данного типа.

При обработке каждой вершины, необходимо добавлять соседей текущей вершины в список ожидания, при этом нужно проверять была ли вершина-сосед уже обработана. Если соседняя вершина еще не обработана, необходимо

посчитать G и H и добавить вершину в обработку, при этом G - дистанция между текущей точкой и соседней, H - дистанция от соседней точки до финиша.

Маской игры будем называть состояние игровых объектов в определенную итерацию сбора объекта. Для работы необходимо описать тип данных отражающий маску игры. В маске должна быть следующая информация:

- Time - время прошедшее от начала игры до текущей итерации.
- Weight - вес маски, определяет приоритет выбора данной маски.
- Mask - массив целых чисел, отражает последовательность сбора объектов, элемент массива это индекс объекта.
- CurrentPoint - индекс последнего объекта маски.

Концептуально алгоритм обработки и создания масок будет похож на A*. Основная работа будет проходить в цикле, список ожидания будет содержать необработанные маски. В процессе обработки маски есть две ветки, первая ветка обрабатывает случай, когда у вершины нет вершин соседей, но персонажу доступны другие вершины для активации, вторая ветка работает только с объектами-соседями.

Алгоритм следует разделить на следующие шаги:

- Определить может ли какой-либо персонаж подобрать ресурс, учитывая типы ресурсов которые способен подбирать данный персонаж.
- Определить скорость подхода и активации, скорость возвращения на точку старта.
- Определить возможность сбора при текущих временных ресурсах.
- Записать значение собранного ресурса и время сбора в список временных ресурсов.
- Записать время окончания действия персонажу.
- Записать новое время в маску.

Маска создается при каждой итерации алгоритма тестировщика, для того чтобы правильно определять текущее время маски, необходимо учитывать время ожидания персонажа.

Чтобы алгоритм мог работать с несколькими финишами потребуется значение выгодности подбора объекта - полезность. Из технического описания процесса игры, можно выделить следующие требования:

- чем ближе объект к квестовму объекту, тем выгоднее его собрать.
- препятствие выгоднее разблокировать, чем собирать ресурс.
- чем быстрее объект можно разблокировать тем он выгоднее.

Значение полезности для препятствия относительно данного квествого объекта будет нормализованное значение глубины вершины, а для ресурса глубина будет увеличена на единицу, чтобы реализовать второе требование(выгоднее разблокировать препятствие, чем собирать ресурс).

```
1 profitability = point.depthStructure(point.
    depthResource+1) / maxDepth
```

Так как квестовых объектов может быть множество, логично предположить, что для расчета полной полезности объекта, нужно сложить полезности относительно всех квествых объектов.

Согласно требованием игры на игровой сцене могут быть несколько персонажей разного и одинакового типа. Необходимо продумать асинхронный сбор ресурсов и структур. Проблема состоит в том, что алгоритм подсчета времени происходит моментально, поэтому нужно определить алгоритм правильного подсчета времени. Данный алгоритм должен поддерживать разные типы персонажей учитывая их скорость перемещения и время затраченное на активацию объектов. Для работы алгоритма потребуется добавить сущность – персонаж(Character). Данная сущность должна быть значимого типа и определять следующие поля:

- Id – уникальный идентификатор персонажа на карте.
- Type – тип персонажа (например "alice "tweedle").
- EndActionTime – время окончания последнего действия персонажа(например персонаж добежит, активирует и добежит обратно в EndActionTime время).

Также потребуется модификация маски игры. Маска должна дополнительно хранить информацию о полученных в определенный момент времени ресурсах. Для этого нужно определить сущность временного

ресурса(TimeResource). Данная сущность должна быть значимого типа и определять следующие поля:

- TimePut – время получения ресурс.
- Type – тип ресурса.
- Count – количество ресурса.

Алгоритм следует разделить на следующие шаги:

- Определить может ли какой-либо персонаж подобрать ресурс, учитывая типы ресурсов которые способен подбирать данный персонаж.
- Определить скорость подхода и активации, скорость возвращения на точку старта.
- Определить возможность сбора при текущих временных ресурсах.
- Записать значение собранного ресурса и время сбора в список временных ресурсов.
- Записать время окончания действия персонажу.
- Записать новое время в маску.

Маска создается при каждой итерации алгоритма тестировщика, для того чтобы правильно определять текущее время маски, необходимо учитывать время ожидания персонажа T_w 1.

$$T_w = T_{ea} - T_m, \tag{1}$$

где T_{ea} - время освобождения персонажа, T_m - текущее время маски. Время ожидания персонажа, для следующей маски, будет рассчитываться по формуле 2.

$$TN_{ea} = T_m + T_w + T_{p1} + T_{p2}, \tag{2}$$

где T_m - текущее время маски, T_{p1} - время за которое персонаж добежит и активирует предмет, T_{p2} - время за которое персонаж вернется на точку старта. Время новой маски рассчитываться по формуле 3.

$$TN_m = T_m + T_w + T_{p1} \tag{3}$$

Вес новой маски рассчитывается по формуле 4.

$$NW_m = W_m + (T_w + T_{p1}) * P, \tag{4}$$

где W_m вес текущей маски, P - полезность текущей точки.

Для хорошего понимания результатов решения задачи, необходимо обрабатывать различные случаи невозможности полноценного вычисления результата, например путь до ключевого объекта может быть заблокирован объектом, на разблокировку которого нехватает ресурсов. В системе должны быть следующие виды контролируемых ошибок:

- Невозможно добраться до объекта - нарушен путь.
- Нет возможности пройти следующий шаг - нехватает ресурсов.
- Нет возможности пройти следующий шаг - ошибка при активации объекта.
- Нет возможности пройти следующий шаг - нет нужного типа персонажа.
- Внутренняя ошибка, включающая System.Exception.

Расчет времени прохождения происходит во время обработки маски игры в конкретной итерации. Каждая следующая активированная вершина прибавляет к общему времени "Time"время, которое требуется для активации вершины. Таким образом, после завершении работы алгоритма общее время прохождение будет время игры самой выгодной маски. **Заключение**. Разработанное программное обеспечение позволяет экономить драгоценное время разработки игры, что в положительную сторону сказывается на бюджете компании. Улучшает качество выпускаемой сборки и позволяет отслеживать проблемы связанные с работой в команде. Позволяет проверять готовые уровни после автоматической сборки.

В данной работе осуществлен анализ способов решения задачи тестирования уровней. Реализовано ядро тестирования и налажено взаимодействие с WonderlandEngine.