УДК 004.932.2

ОЦЕНКА ПЕРСПЕКТИВНОСТИ ГИБРИДНОГО МОДЕЛИРОВАНИЯ ВИЗУАЛЬНОГО ВОСПРИЯТИЯ ИГРОВЫХ АГЕНТОВ С ИСПОЛЬЗОВАНИЕМ ОТРИСОВКИ ТЕКСТУР

А. М. Примаченко[0009-0008-6396-2035]

Казанский (Приволжский) федеральный университет, г. Казань, Россия

primachenko.artem@mail.ru

Аннотация

Представлен алгоритм системы визуального восприятия для игровых агентов, реализованный в игровом движке Unity. Предложенный метод основан на сравнении изображений с двух камер, учитывающих сложные визуальные эффекты (освещение, тени, маскировку), и дополнен проверкой прямой видимости, учетом скорости движения объекта и механикой постепенного обнаружения. Алгоритм оптимизирован с применением асинхронных вычислений, динамической активации камер и ускоренных алгоритмов. Выполнено тестирование системы при различных уровнях нагрузки, сделаны выводы об условиях, оптимальных для работы алгоритма. Проведен также анализ научной литературы по схожим решениям, выявлены их сильные и слабые стороны. Результаты могут быть применены в разработке видеоигр для создания реалистичного поведения неигровых персонажей, особенно в играх с элементами скрытности.

Ключевые слова: *видеоигры, искусственный интеллект, система восприятия, NPC, Unity, компьютерное зрение, геймдизайн.*

ВВЕДЕНИЕ

Системы визуального восприятия неигровых персонажей (англ. non-playable character, NPC) в видеоиграх являются важным элементом, обеспечивающим реалистичность и интерактивность игрового процесса. Как отмечено в ряде работ [1-11], информация о видимости (или линии обзора) является ключевым элементом, позволяющим NPC строить пространственное представление о мире игры и принимать решения в реальном времени.

Традиционные подходы (проверка углов обзора, использование рейкастинга (от англ. raycast) и др.), обладают ограничениями, связанными с недостаточным учетом визуальных факторов, таких как освещение, тени и маскировка. Это приводит к неестественному поведению NPC, особенно в играх, использующих игровые механики невидимости («стелс-играх», от англ. stealth), где требуется высокая степень реализма [1].

Наиболее распространенным способом реализации системы зрения для неигровых персонажей является алгоритм из двух шагов: проверка нахождения целевого объекта в определенной зоне в игровом мире, часто динамически изменяющейся в зависимости от положения персонажа и называемой полем зрения, и проверка на наличие препятствий между персонажем и целевым объектом. Хотя такой подход прост в реализации, он не учитывает множество факторов реального визуального восприятия, что снижает убедительность игрового опыта.

Цель настоящего исследования была разработка системы визуального восприятия, которая учитывает сложные визуальные эффекты, обеспечивает гибкость настройки и в то же время остается оптимизированной для реального времени. Объектом исследования выступали методы реализации визуального восприятия NPC, а предметом – алгоритм, основанный на сравнении изображений и дополнительных механиках.

В работе [12] нами были проанализированы существующие системы визуального восприятия, выявлены их сильные и слабые стороны, а также был представлен спроектированный алгоритм и предложены методы его оптимизации и тестирования. В настоящей работе описаны детали его реализации, включая оптимизацию, а также процесс и результаты тестирования.

ПРОЕКТИРОВАНИЕ

Для реализации системы зрения игрового агента необходимо определить, имеет ли наблюдатель прямую видимость объекта. Наиболее распространённым способом организовать данную механику является проверка пересечения объектом поля зрения наблюдателя [1-8], а также проверка наличия препятствий между наблюдателем и объектом при помощи рейкастов (англ. raycast) [1-9, 13]. Главная проблема этого подхода: он не учитывает

сложные визуальные эффекты, такие как освещение, тени, отражения и маскировка. Поэтому требуется более точный и гибкий подход, который мог бы учитывать визуальные особенности сцены.

Основной алгоритм

Проблема: традиционные методы определения видимости не учитывают визуальные эффекты, такие как освещение, тени и маскировка, что снижает реалистичность поведения NPC.

Решение: основной алгоритм системы зрения основан на сравнении изображений, отрисованных в игровом движке с точки зрения наблюдателя. Первое изображение включает в себя вид сцены с видимой целью, а второе – сцены без цели. Помимо цели оба изображения отрисовывают те же элементы сцены, что и основная камера игрока, включая уровень и освещение. Это позволяет учитывать все визуальные эффекты, такие как тени и отражения (рис. 1).

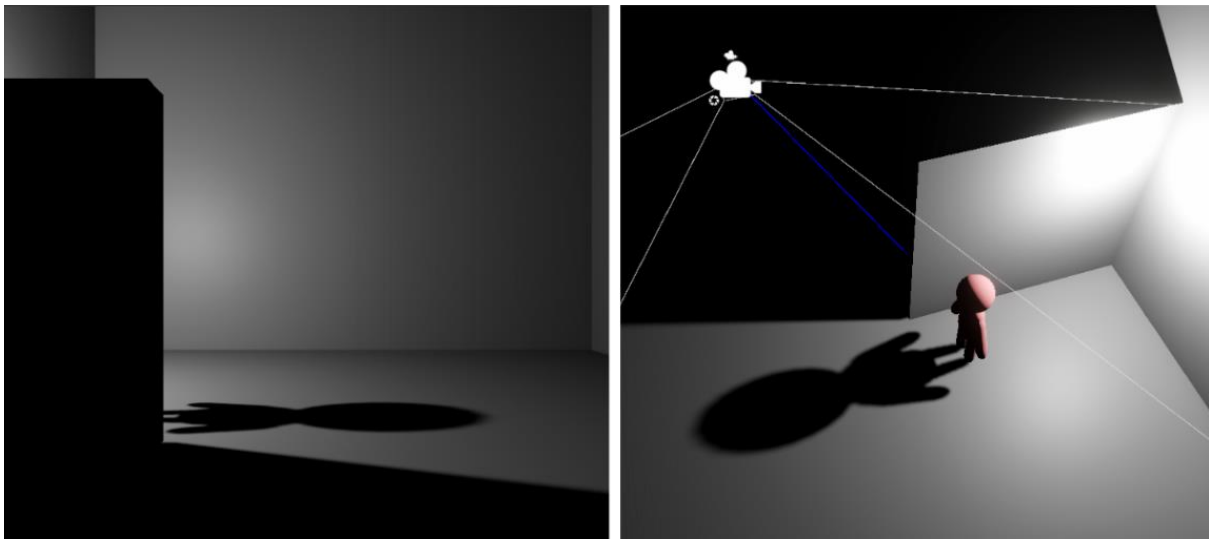

Рис. 1. Наблюдатель видит тень цели, но не саму цель.

Шаги решения:

1. Рендеринг сцены, включающей объект цели, в формат изображения.
2. Рендеринг сцены, исключающей объект цели, в формат изображения.
3. Сравнение пикселей, состоящее в вычислении разности пикселей по каналам RGB и суммировании для получения степени расхождения двух

изображений D. Для каждой пары соответствующих пикселей текстур вычисляем разность по каналам RGB. Эти разности суммируем, чтобы получить значение D:

$$D = \frac{\sum_{i=1}^{N} (|R_1 - R_2| + |G_1 - G_2| + |B_1 - B_2|)}{3\ N},$$

где R_1, G_1, B_1 – значения каналов RGB для i-го пикселя на первом изображении, R_2, G_2, B_2 – значения каналов RGB для i-го пикселя на втором изображении, N – общее количество пикселей одного изображения.

4. Расчет видимости. По значению D с помощью некоторой функции рассчитаем величину V видимости цели. Если V превышает пороговое значение V_{min}, считаем, что наблюдатель видит цель, т. е. если $V > V_{min}$, то цель видима. Функция преобразования D в V может, например, следовать закону убывающей доходности

$$V = V_{max} D/(D + a),$$

где V_{max} – максимально возможное значение V, a – положительный коэффициент убывающей доходности; чем меньше значение a, тем быстрее V достигает значения, близкого к V_{max}, и выходит на «плато».

Подобный подход заимствует методы компьютерного зрения, попытки применить которые к компьютерным играм предпринимались ранее [14], однако в данном случае также используется преимущество дополнительной информации об искомом объекте, которую алгоритм может получить напрямую от игрового движка.

Такая реализация учитывает визуальные особенности сцены (освещение тени), а также предоставляет гибкость в виде множества переменных для настройки под конкретные обстоятельства, однако обладает высокой вычислительной сложностью из-за необходимости рендерить сцену несколько раз и считать расхождение изображений.

Периферийное зрение

Проблема: объекты, на которые направлен взгляд наблюдателя, являются настолько же заметными с точки зрения системы, как и объекты на границе его периферийного зрения, поскольку пиксели в центре созданного изображения имеют такой же визуальный вес, как и пиксели вдали от центра.

Решение: используем двоичную маску, чтобы определить, какие пиксели изображения стоит сравнивать, а какие – игнорировать. Чем дальше от центра маски, тем меньшее число пикселей учитываем при расчете D.

Таким образом, объекты на краю периферии наблюдателя оказывают влияние на меньшее число весомых отрисованных пикселей, значит, дают меньшее значение D, при этом цели по направлению взгляда наблюдателя остаются настолько же заметными. Кроме того, уменьшается вычислительная сложность алгоритма за счет меньшего объема необходимых вычислений разности пикселей, но увеличивается риск пропустить некоторые детали.

Маскировка объектов

Проблема: изначальный алгоритм сравнивает отдельные пиксели, что делает его чрезмерно чувствительным к мелким деталям; например, если цель – это персонаж, одетый в лесной камуфляж на фоне лесной сцены, даже небольшие различия в текстуре могут привести к его обнаружению. Это не позволяет объекту «сливаться» с окружающей обстановкой.

Решение: для уменьшения чувствительности к мелким деталям применяют размытие по Гауссу, которое снижает резкость изображения, что позволяет объекту маскироваться. Такой подход является также хорошим приближением того, как работает зрительное восприятие человека в реальности [15]. Сложность алгоритма размытия по Гауссу составляет $O(nr^2)$, где n – количество пикселей, а r – радиус круга, по которому усредняется каждый пиксель (другими словами, сила размытия).

Вместо традиционного гауссова фильтра можно применить фильтр Box Blur, который выполняет простое усреднение значений пикселей в квадратном окне. При нескольких последовательных пропусках Box Blur позволяет достичь результата, практически не отличимого от размытия по Гауссу, имея при этом линейную вычислительную сложность $O(n)$, что значительно снижает затраты по сравнению с гауссовым фильтром [16].

Шаги решения:

1. Размытие по Гауссу: перед сравнением текстур применим размытие по Гауссу. Это уменьшит резкость изображения и снизит чувствительность

алгоритма к мелким деталям. Радиус определяет степень размытия и обычно устанавливается в диапазоне от 1 до 5 пикселей.

2. Среднеквадратичная ошибка (MSE): вместо простой суммы разностей используем среднеквадратичную ошибку для расчета V:

$$V = \frac{\sum_{i=1}^{N} ((R_1 - R_2)^2 + (G_1 - G_2)^2 + (B_1 - B_2)^2)}{3\,N},$$

что позволит уменьшить влияние незначительных изменений в поле зрения и увеличить вес серьезных различий.

В итоге уменьшится чувствительность наблюдателя к мелким деталям и шуму, а у цели появится возможность маскироваться в подходящей обстановке, однако еще больше увеличится вычислительная сложность алгоритма.

Проверка прямой видимости

Проблема: наблюдатель может реагировать на тени, отражения или малую часть объекта так же, как и на сам объект. Это приводит к ложным срабатываниям, когда наблюдатель «видит» цель, хотя на самом деле он видит только ее тень или отражение.

Решение: для устранения этой проблемы введем проверку рейкастами. На модели цели равномерно распределим *X* точек, которые проверим на видимость с помощью рейкастов (рис. 2).

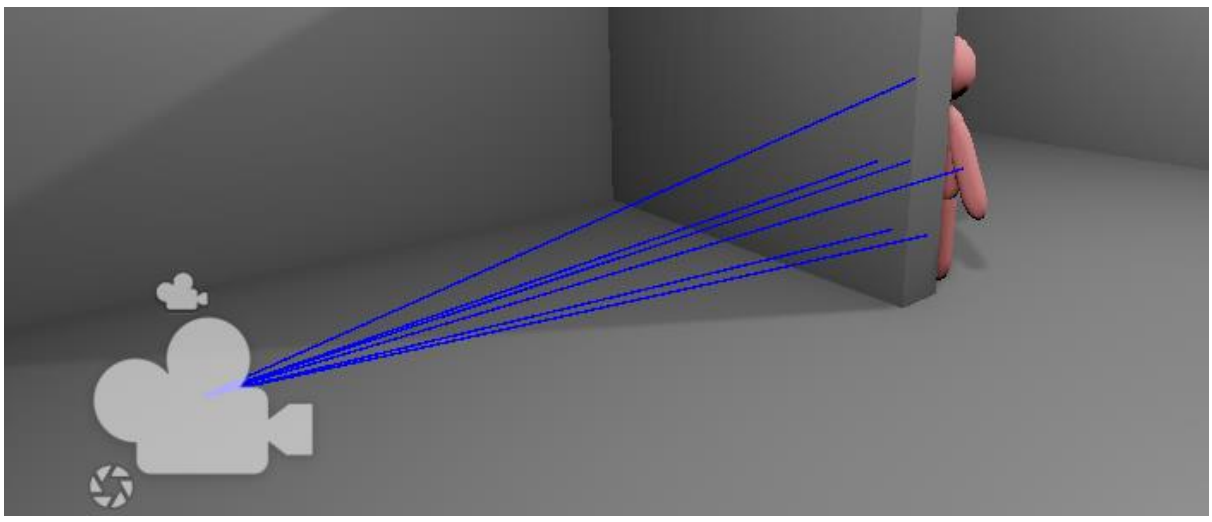

Рис. 2. Пучок из 6 рейкастов, направленных в разные части цели.

Если количество точек X_{pass}, прошедших проверку, превысит пороговое значение X_{min}, считаем, что наблюдатель видит сам объект. Если $X_{pass} < X_{min}$, то наблюдатель видит тень, отражение или малую часть объекта. В этом случае наблюдатель получит информацию о координатах цели и с помощью отдельной логики попытается получить визуальное подтверждение объекта.

Шаги решения:

1. Распределение точек на цели: на модели цели равномерно распределим X точек, где X обычно принимает значения от 5 до 20 в зависимости от размера и сложности модели.

2. Проверка рейкастами: для каждой точки выполним рейкаст от наблюдателя до точки. Если луч не пересечет препятствий, точку считаем видимой.

3. Определение видимости цели: если количество видимых точек X_{pass} превышает пороговое значение X_{min}, цель считаем видимой. В противном случае наблюдатель видит только тень или отражение.

4. Попытка визуального подтверждения: если наблюдатель обнаружил объект, но не имеет прямой видимости, он попытается получить визуальное подтверждение объекта, например, повернуться в его сторону или же при помощи алгоритма поиска пути дойти до позиции объекта.

Таким образом, можно разделить логику поведения наблюдателя, когда он имеет прямую видимость цели или видит «подсказку», где она находится. Это позволит сделать поведения наблюдателя более реалистичным, но еще больше увеличит вычислительную сложность алгоритма из-за необходимости выполнения рейкастов.

Влияние скорости движения объекта на его заметность

Проблема: в реальности быстро движущиеся объекты заметить легче, чем медленно движущиеся или статичные. Текущий алгоритм не учитывает скорость движения цели, что может привести к нереалистичному поведению наблюдателя.

Решение: для учета скорости движения цели введем параметр eV – эффективное значение V, который рассчитаем как произведение V на

множитель, зависящий от скорости движения цели. Чем выше скорость цели, тем больше множитель, что увеличит заметность объекта.

Шаги решения:

1. Расчет скорости цели: определим скорость движения цели на основе ее позиции в предыдущих кадрах.
2. Расчет множителя: множитель α для V рассчитаем на основе скорости цели. Например, множитель может быть линейно зависимым от скорости.
3. Расчет V: эффективное значение V рассчитываем как $e \cdot V = V \cdot \alpha$.
4. Если $e \cdot V > V_{min}$, значит, наблюдатель видит цель.

Такая реализация обладает большей реалистичностью и создает дополнительный геймплейный элемент для стелс-игр, где игрок должен двигаться медленно, чтобы оставаться незамеченным, но может потребовать осторожной настройки вычисления значения α.

Механика постепенного обнаружения

Проблема: в некоторых случаях обнаружение не должно быть мгновенным. Например, наблюдатель может постепенно «замечать» цель, что добавит реализма и в случае стелс-игр сделает геймплей более честным с точки зрения игрока в ситуации, когда наблюдателями являются противники, а объектом – сам игрок.

Решение: для реализации постепенного обнаружения введем параметр Det – степень обнаружения, который изменяется в зависимости от значения V и скорости обнаружения Det_{speed}.

Если $V > V_{min}$, то Det увеличится на ΔDet Det_{speed} в каждом кадре. Если $V < V_{min}$, то Det уменьшится на величину угасания Dec. Когда Det превысит пороговое значение Det_{min}, цель считаем обнаруженной.

Это дает системе еще большую гибкость и в случае стелс-игр создает более честный игровой опыт с точки зрения игрока, давая ему возможность среагировать на угрозу, однако может потребоваться дополнительная логика для реакции наблюдателя на «подозрительный» объект, когда $0 < Det < Det_{min}$.

Оптимизация

Рассматриваемый алгоритм имеет высокую вычислительную сложность, что может негативно сказаться на производительности игры, особенно при большом количестве наблюдателей и целей. Для повышения производительности предложим следующие оптимизации.

1. Асинхронная реализация: системе необязательно выполнять все необходимые расчеты за один игровой кадр, вместо этого алгоритм выполнится в побочных потоках программы, а наблюдатель среагирует на изменения, как только закончится выполнение алгоритма.
2. Отрисовка сцены: происходит только непосредственно перед расчетом значения D, а не в каждом кадре, исключая лишнюю нагрузку на видеокарту.
3. Распределение нагрузки: в случае существования на сцене нескольких наблюдателей их обработка происходит по алгоритму Round-Robin, что увеличивает время реакции каждого из наблюдателей, но также снижает максимальную нагрузку на процессор и видеокарту.
4. Снижение разрешения текстур: для расчета D используем текстуры с более низким разрешением, чем основная камера игры, что значительно уменьшит количество пикселей для обработки.
5. Параллелизация: некоторые части алгоритма, например размытие по Гауссу, могут выполняться в параллельном режиме, задействуя несколько ядер процессора и тем самым снижая время выполнения задачи.

РЕАЛИЗАЦИЯ

В качестве платформы для реализации системы зрения был выбран игровой движок Unity, что обусловлено такими его преимуществами, как встроенные инструменты рендеринга для работы с камерами и текстурами, удобная система физики для точных рейкаст-проверок и возможность быстрого прототипирования с визуальной отладкой.

Компоненты системы

Реализованная система состоит из трех основных компонентов:

- наблюдаемых объектов (Observable);
- объектов-наблюдателей (Observer);

- класса, управляющего наблюдателями (VisionSystem),

а также трех вспомогательных компонентов:

- для наблюдаемых объектов – это компонент, реализующий проверку линий видимости (RaycastClusterCheck);

- для объектов наблюдателей – генерация маски для сравнения текстур (VisionMask);

- для управляющего класса – опциональный модуль размытия текстур (FastGaussianBlur).

Компонент Observable реализует класс наблюдаемых объектов, т. е. объектов, имеющих визуальное представление на сцене и распознаваемых наблюдателями. Он содержит информацию о компонентах отрисовки, методы для управления слоями этих компонентов, а также информацию о скорости объекта и настройки проверки линий видимости.

С каждым объектом класса Observable связан объект вспомогательного класса RaycastClusterCheck, который определяет на цели точки в пространстве, относительно которых будет проходить проверка линий видимости, а также реализует логику этой самой проверки.

Компонент Observer реализует класс объектов наблюдателей. Каждый наблюдатель может отслеживать одну или несколько целей. В классе хранятся пары отрисованных текстур поля зрения наблюдателя, включающие и исключающие обозначенные цели, а также реализованы методы отрисовки или же обновления этих текстур. После каждого обновления текстур вызывается событие TexturesWereUpdated, чтобы уведомить любые зависимые классы, например UIManager, о необходимости их собственного обновления.

Кроме того, в классе Observer определены механики обнаружения целей:

- показатель расхождения текстур преобразуется в значение видимости целей;
- в зависимости от этого значения растет или уменьшается показатель подозрения наблюдателя;
- при достаточно высоком значении показателя подозрения происходит переход в режим тревоги.

Отметим, что описанная логика реализуется только в том случае, если проверка линий видимости отслеживаемых объектов-целей оказалась успешной, в соответствии с выставленными параметрами этих объектов.

С каждым объектом класса Observer связан вспомогательный объект класса VisionMask, который в начале работы программы генерирует и сохраняет маску зрения наблюдателя, на которую будет ссылаться класс, вычисляющий расхождение текстур. Маска определяет, какие пиксели будут сравниваться, а какие игнорироваться. Маска является максимально плотной в центре, что соответствует сравнению каждого включенного пикселя, и быстро теряет плотность по мере удаления от центра.

Класс VisionSystem является ключевой компонентой, координирующей работу системы зрения. Он управляет обновлением текстур наблюдателей, которое в целях оптимизации происходит по алгоритму Round-Robin. Здесь также реализованы алгоритмы сравнения текстур поля зрения, хранящихся у наблюдателей, для подсчета значения их расхождения, а также опциональная генерация тепловой карты, используемой в процессе тестирования системы. Все методы реализованы в асинхронном режиме, что позволяет системе работать, не задерживая основной поток программы.

Если переменная VisionBlurRadius объекта класса Observer имеет ненулевое значение, перед сравнением текстур происходит их обработка при помощи наложения размытия по Гауссу, реализованная в классе FastGaussianBlur. Значение VisionBlurRadius в этом случае берется в качестве параметра радиуса или же степени размытия. Для обработки текстур использован специальный алгоритм, выдающий результат, практически не отличимый от традиционного гауссова размытия, но обладающий вычислительной сложностью O(N) вместо стандартной O(NR), где N – количество пикселей в изображении, а R – радиус размытия.

Кроме перечисленных компонентов существуют классы PlayerControl и UIManager, не являющиеся частью системы зрения, но используемые в процессе тестирования системы. PlayerControl обеспечивает функции управления передвижением персонажа игрока и направления взгляда его камеры. UIManager сохраняет и выводит в интерфейс игры актуальную информацию о работе системы зрения, включая вычисляемые параметры наблюдателей

(например, расхождение текстур или уровень подозрения), время реакции наблюдателей, отрисованные или обработанные текстуры, тепловую карту, значение FPS (количество кадров видеоигры, отрисовываемых за секунду, от англ. frames per second) и пр.

На рис. 3 представлена изображена диаграмма классов, визуализирующая связь объектов в системе. Отметим, что на диаграмме изображены лишь классы, формирующие собой систему зрения, что не относится к классам PlayerControl и UIManager. Диаграмма также не включает в себя встроенные классы Unity.

Далее дано описание всех перечисленных классов и деталей их работы сказано ниже.

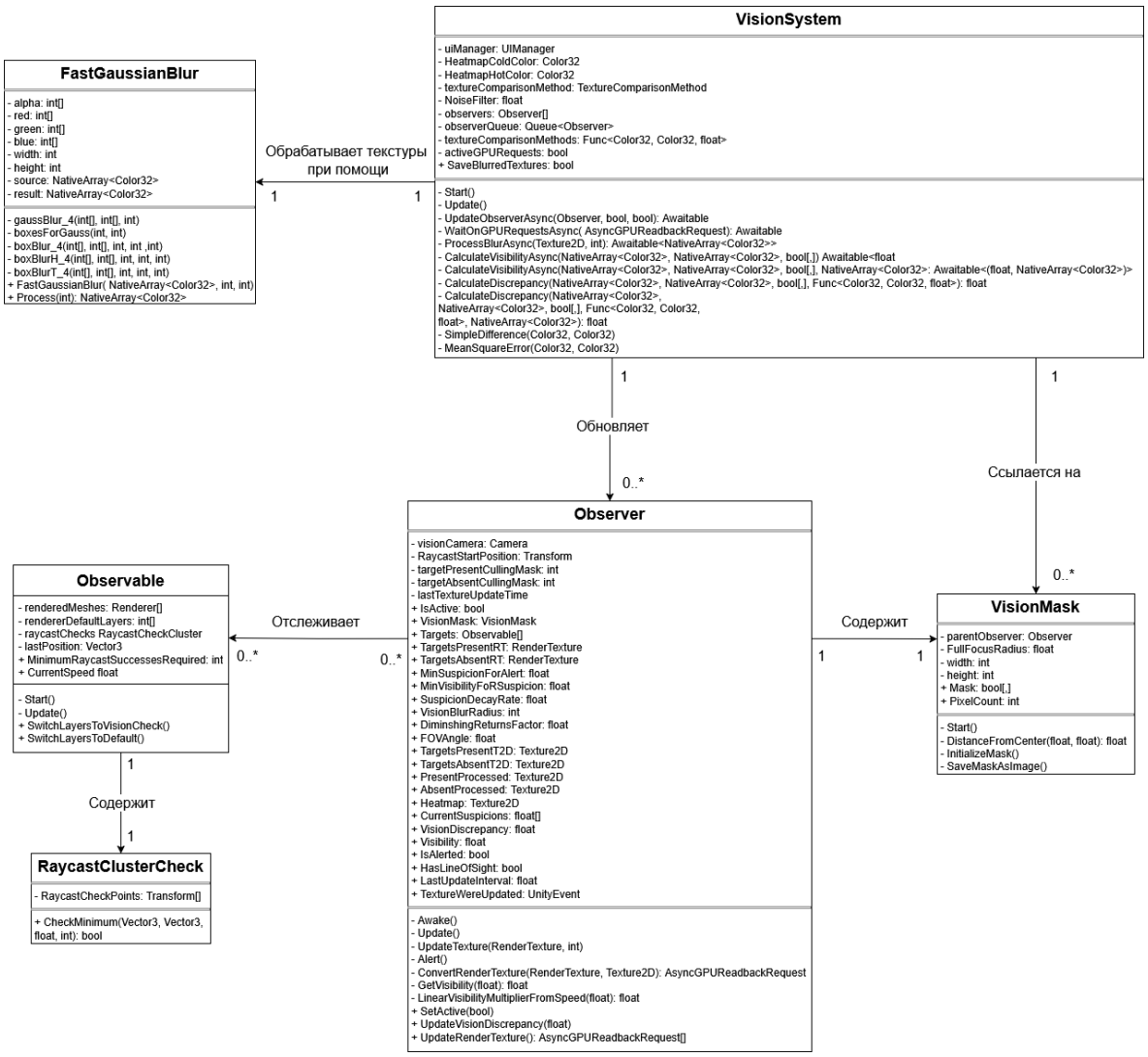

Рис. 3. Диаграмма классов, описывающая систему зрения.

Observable

Компонент Observable реализует класс наблюдаемых объектов, т. е. объектов, имеющих визуальное представление на сцене и распознаваемых наблюдателями, объектами класса Observer.

Для корректной работы наблюдаемому объекту необходимо предоставить RenderedMeshes – массив ссылок на объекты класса Renderer, т. е. не объекты, отвечающие за отрисовку визуального отображения моделей на камерах, присутствующих на сцене. Другими словами, это все составные части игрового объекта, по которым его можно визуально идентифицировать.

В классе Observable реализованы вспомогательные методы SwitchLayersToVisionCheck() и SwitchLayersToDefault(), исполняющие функцию переключения «слоев» объектов из массива RenderedMeshes – эта функция используется позднее в скрипте Observer.

Для достижения корректного переключения объектов между слоями в начале работы скрипта слои всех объектов в RenderedMeshes сохраняем в массив слоев по умолчанию – rendererDefaultLayers, чтобы далее можно было после любой работы со слоями вернуть их в начальное состояние.

Метод SwitchLayersToVisionCheck() изменяет слой каждого из объектов в renderedMeshes на созданный слой VisionCheck, который будет игнорироваться во время рендера текстуры, исключающей искомые цели наблюдателя. Метод SwitchLayersToDefault() также изменяет слои объектов renderedMeshes, однако он возвращает их к сохраненным значениям по умолчанию.

Кроме того, поскольку система зрения учитывает скорость, с которой движется объект-цель, класс Observable в каждом кадре рассчитывает скорость родительского объекта. Достигается это путем сохранения вектора – предыдущей позиции объекта – через компонент Transform и вычисление разности с текущей позицией. Длина вектора, полученного таким вычитанием, делится на время, прошедшее с выполнения предыдущего кадра (Time.deltaTime), чтобы получить мгновенную скорость.

Каждый объект класса Observable также обязан быть связан с соответствующим объектом класса RaycastCheckCluster, описанного ниже.

RaycastCheckCluster

Класс RaycastCheckCluster является вспомогательным компонентом для класса Observable и реализует логику проверки линий видимости объекта относительно заданной позиции.

Для корректной работы класса сначала необходимо создать набор пустых объектов с компонентом Transform, которые будут представлять собой точки в пространстве – позиции, на которые будут направлены рейкасты (рис. 4).

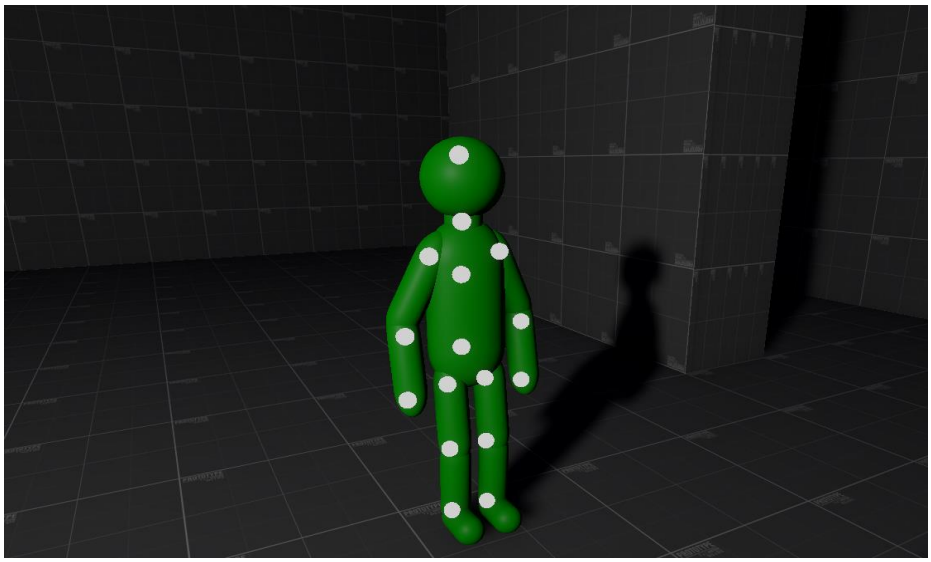

Рис. 4. Расставленные на модели точки проверки линий видимости.

Логика проверки линий видимости реализована в методе CheckMinimum(), который возвращает успех или неудачу проведенной проверки. В нем происходит проверка рейкастом для каждой из точек в массиве RaycastCheckPoints: луч выпускается из позиции raycastStart в направлении точки, и если он успешно достигает ее, не встретив препятствий, и при этом угол между вектором рейкаста и forwardVector не больше fov, то счетчик успехов увеличивается. Если количество успешных проверок достигает значения minSuccesses, то метод возвращает «истину».

Observer

Компонент Observer реализует класс объектов-наблюдателей и включает логику отрисовки текстур поля зрения, вызова проверки линий видимости, расчета показателя видимости цели и управления уровнем подозрения наблюдателя.

Observer – первичная настройка

У каждого наблюдателя должна быть определена ссылка на компонент Camera – камеру, которую наблюдатель будет использовать для отрисовки поля зрения. В практическом смысле камера – это глаза наблюдателя. По этой причине для наибольшей реалистичности рекомендуется расположить эту камеру там же, где на игровой модели персонажа расположены глаза (например, в случае, если наблюдатель – камера видеонаблюдения, там, где расположен объектив), повернуть ее так, чтобы вектор направления «вперед» совпал с ожидаемым направлением взгляда персонажа, а вектор «вниз» в большинстве случаев рекомендуется установить по направлению гравитации в игровом мире. Камеру также рекомендуется сделать дочерним объектом модели персонажа, чтобы она предсказуемым образом повторяла все его перемещения и повороты.

Изменение параметров компонента камеры позволяет влиять на то, что будет видеть NPC во время игрового процесса. Существующие настройки включают угол обзора камеры, минимальную и максимальную дистанции геометрии от камеры для ее отрисовки (англ. Clipping planes), маску слоев, отрисовываемых камерой, и т. д.

Для корректной работы проверки линий видимости между наблюдателем и целью системе необходимо иметь информацию об установленном угле обзора камеры. По умолчанию компонент Camera в Unity позволяет получить информацию только о вертикальном угле ее обзора.

Горизонтальный угол обзора рассчитываем по следующей формуле

$$hFOV = \frac{180}{\pi} 2arctg(aspect\ tg(vFOV \frac{\pi}{180} 0.5)),$$

где aspect – соотношение сторон ширины и высоты камеры, а $vFOV$ – ее вертикальный угол обзора. В качестве значения FOVAngle выбираем наибольшую из величин $vFOV$ и hFOV, чтобы убедиться, что объекты, находящиеся за пределами меньшего угла обзора, но в пределах большего, будут успешно обнаружены.

Observer – отрисовка текстур

Для корректной работы наблюдателя необходимо создать в ассетах проекта текстуру класса RenderTexture и назначить ее в поле RenderTextureSettings, которое будет использовано как образец для текстур отрисовки выбранного наблюдателя. В параметрах созданной RenderTexture можно задать такие настройки, как разрешение текстуры, параметры сглаживания и фильтрации, а также формат цвета текстуры. Для различных наблюдателей можно создавать разные образцовые ассеты RenderTexture или же использовать одну и ту же текстуру для всех.

При назначении разрешения текстуры рекомендуется сохранять то же соотношение сторон, что использует камера выбранного наблюдателя. Конкретные значения ширины и высоты текстуры рекомендуется сохранять в пределах нескольких сотен пикселей для наилучшей оптимизации. Рекомендуемый формат цвета – это R8G8B8A8_UNORM, с которым система работает лучше всего.

При начале работы системы каждый компонент Observer создает два объекта RenderTexture, копируя настройки, указанные в RenderTextureSettings: TargetsPresentRT и TargetsAbsentRT. Первая будет использована для сохранения результатов отрисовки, включающей объекты-цели, а вторая – для отрисовки, исключающей их. Создаются также маски отбраковки (англ. Culling mask) targetsPresentCullingMask и targetsAbsentCullingMask для использования при отрисовке текстур TargetsPresentRT и TargetsAbsentRT.

Для отрисовки текстур реализован метод UpdateRenderTextures(), однако внутри класса Observer он не вызывается – вместо этого его вызов происходит в нужное время из управляющего наблюдателями класса VisionSystem. Первое, что необходимо сделать перед отрисовкой текстур, – переключить слои объектов Renderer у объектов целей на слой VisionCheck. Для этого у каждой из целей в массиве Targets происходит вызов метода SwitchLayersToVisionCheck(). После этого выполним следующий алгоритм.

1. Вызовем метод для обновления текстур UpdateTexture(). В нем у камеры наблюдателя изменится целевая текстура, т. е. текстура, в которую будет записан результат отрисовки, а также назначена ее маска отбраковки. Затем

происходит ручной вызов Render() метода отрисовки, определенного в классе Camera, который позволит произвести отрисовку изображения с камеры в целевую текстуру при использовании назначенной маски отбраковки. В конце целевая текстура камеры изменится на значение null, чтобы избежать неожиданных изменений в ранее назначенной текстуре.

2. Вызовем метод ConvertRenderTexture(), асинхронно преобразующий текстуру класса RenderTexture в текстуру класса Texture2D для последующей обработки и расчета несоответствия в классе VisionSystem.

3. В методе ConvertRenderTexture() выполним следующий код (Листинг 1). Класс AsyncGPUReadback позволит асинхронным образом прочитать информацию о записанной в память видеокарты текстуры renderTexture. Эта информация затем будет использована, чтобы скопировать данные о пикселях renderTexture и записать их в текстуру targetTexture, также асинхронно, сохранив таким образом изображение в новом формате. Класс возвращает объект асинхронного запроса, который можно использовать для проверки завершения процесса записи данных пикселей. По завершению выполнения запроса в текстуре TargetsPresentT2D сохранится обновленное изображение камеры наблюдателя, включающее в себя отрисовку объектов-целей (рис. 5).

Листинг 1. Асинхронное преобразование текстуры.

```
    var request = AsyncGPUReadback.Request(renderTexture, 0,
(AsyncGPUReadbackRequest asyncAction) =>
{
    targetTexture.SetPixelData(asyncAction.GetData<byte>(), 0);
    targetTexture.Apply();
});
return request;
```

Все перечисленные шаги, применяемые к объектам TargetsPresentRT, targetsPresentCullingMask и TargetsPresentT2D, повторяются и для объектов TargetsAbsentRT, targetsAbsentCullingMask и TargetsAbsentT2D, чтобы получить изображение текстуры, исключающей отрисовку объектов-целей (рис. 5).

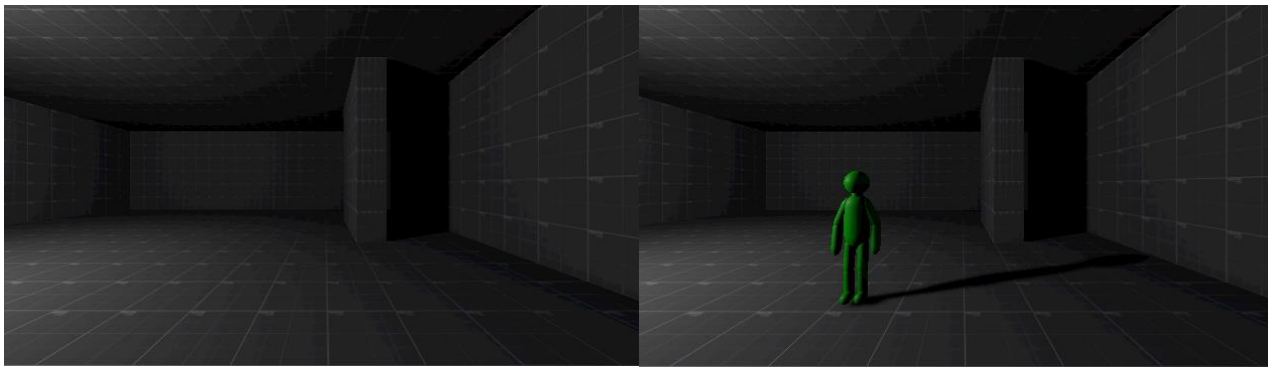

Рис. 5. Пример текстуры, включающей цель (слева), и текстуры, исключающей цель (справа).

После выполнения всех этих операций для всех объектов в массиве Targets вызываем метод SwitchLayersToDefault(), чтобы вернуть компонентам Renderer у целей их слои по умолчанию и позволить им продолжить работу в привычном режиме.

Observer – механики подозрения

Финальный результат работы системы зрения состоит в изменении уровня подозрения у всех наблюдателей. Происходит это исходя из рассчитанного значения несоответствия текстур (VisionDiscrepancy).

Значение VisionDiscrepancy изменяется извне класса путем вызова метода UpdateVisionDiscrepancy() классом VisionSystem. Этот метод устанавливает новое значение несоответствия текстур, а также новое значение величины видимости объектов в поле зрения, рассчитанное в методе GetVisibility(). Метод использует следующую формулу

$$V = 100D \div (D + 0.1^{P}),$$

где *V* – видимость, *D* – значение несоответствия текстур, а *P* – значение параметра VisibilityDiminishingReturnsPower.

В методе Update() в каждом кадре работы программы происходит изменение уровней подозрения наблюдателя в зависимости от рассчитанного значения Visibility. Для каждой из целей наблюдателя реализован следующий алгоритм.

1. Рассчитаем значение эффективной видимости по формуле

$$eV = V(1 + \frac{speed}{2}),$$

где *V* – значение Visibility, *eV* – эффективная видимость цели, а speed – ее скорость.

2. Проведем проверку линий видимости между наблюдателем и целью путем вызова метода CheckMinimum() у привязанного к цели объекта класса RaycastCheckCluster. Минимальное число успехов – это настраиваемый параметр у выбранной цели.

3. Если значение эффективной видимости равно или превышает значение параметра MinVisibilityForSuspicion, значение уровня подозрения для текущей цели увеличим на величину, равную произведению эффективной видимости и времени, прошедшего с момента последнего вызова метода Update() (Time.deltaTime).

4. Если уровень подозрения цели достигнет значения MinSuspicionForAlert, то вызовем метод Alert(), принимающий текущую цель и результат проверки линий видимости, в зависимости от которого может выполняться разная логика, конкретная реализация которой не входит в рамки системы зрения.

5. Если же значение эффективной видимости меньше значения параметра MinVisibilityForSuspicion, то значение подозрения для текущей цели уменьшим на величину, равную произведению параметра SuspicionDecayRate и значения Time.deltaTime.

VisionMask

У каждого объекта-наблюдателя должен быть определен компонент VisionMask, который отвечает за генерацию и хранение маски зрения – массива Mask, передающего системе зрения информацию о том, какие пиксели текстур нужно сравнивать, а какие – игнорировать. Двумерная маска представляет собой массив булевых значений, где значение «истина» в ячейке (*i*; *j*) сообщает о необходимости принять соответствующие пиксели на позициях (*i*; *j*) в расчет величины несоответствия текстур, а значение «ложь» – о необходимости проигнорировать их.

Инициализированная маска сообщает системе о необходимости сравнивать каждый пиксель текстур в некотором настраиваемом радиусе от их

центра, а вне этого радиуса плотность сравниваемых пикселей уменьшается в кубической зависимости от расстояния пикселя до центра (рис. 6).

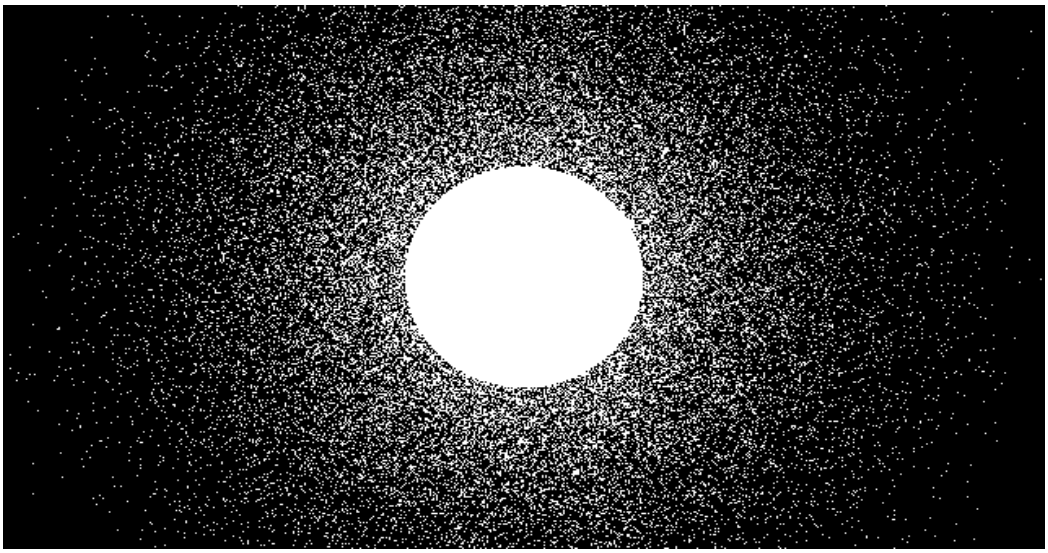

Рис. 6. Визуализация маски зрения.

Использование маски имеет сразу несколько преимуществ:

- имитация периферийного зрения – участки текстур вблизи ее центра имеют намного больший вес в сравнении с участками далеко от центра, что соответствует поведению, когда объекты по направлению взгляда наблюдателя являются значительно более заметными по сравнению с объектами на грани периферии;
- оптимизация – поскольку количество пикселей, для которых происходит подсчет разностей, резко уменьшается по мере удаления от центра текстур, также снизится и общее количество необходимых вычислений;
- открыта возможность использования более высоких разрешений отрисовываемых и сравниваемых текстур, поскольку вычисления разностей малозначимых пикселей на краю периферийного зрения больше не оказывают влияние на производительность, а точность вычислений в центре значительно увеличивается. При этом стоит отметить, что маска не влияет на время отрисовки текстур, а влияет лишь на время их сравнения.

FastGaussianBlur

Для выполнения функции управления зоркостью наблюдателей в систему был интегрирован опциональный модуль, реализующий обработку текстур при

помощи размытия по Гауссу. Использование этого модуля определяется параметром VisionBlurRadius класса Observer, а исполняется он в классе VisionSystem.

Модуль принимает в качестве параметра массив из объектов Color32, являющийся представлением текстуры в формате NativeArray, который позволяет работать с памятью, хранящей в себе текстуру напрямую.

Процесс размытия выполняется только при вызове метода Process(), принимающего в качестве параметра целочисленный радиус гауссова размытия. Метод формирует новые массивы цветовых каналов, которые будут отражать цвета пикселей новой, размытой текстуры. Алгоритм основан на наблюдении о том, что применение несколько раз размытия “Box Blur” к одному и тому же изображению дает результат, приближенный к гауссовому размытию [16]. Классический алгоритм размытия по Гауссу имеет вычислительную сложность $O(nr^2)$, где *n* – количество пикселей в изображении, а *r* – радиус размытия. В то же время оптимизированный алгоритм имеет вычислительную сложность $O(n)$, позволяя значительным образом снизить время вычислений, при этом погрешность результата в сравнении с классическим алгоритмом составляет в среднем всего 0.04% на пиксель [17].

Значения цветовых каналов, полученные в результате выполнения алгоритма, сохраняются в массиве цветов формата Color32, являющемся представлением результата обработки текстуры гауссовым размытием (рис. 7). Итоговый массив возвращается методом Process().

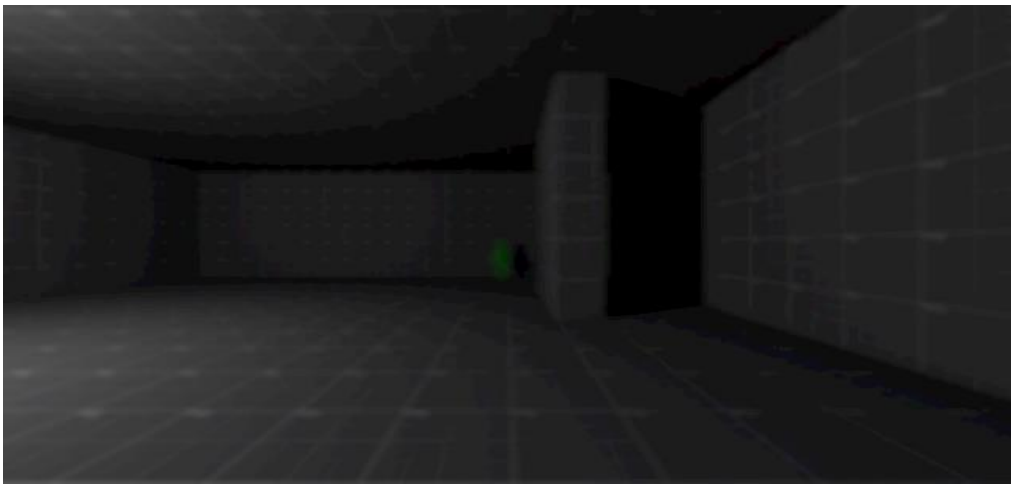

Рис. 7. Пример текстуры, обработанной гауссовым размытием.

VisionSystem

Класс VisionSystem координирует работу системы зрения путем вызова методов обновления текстур отрисовки наблюдателей, обработки текстур, сравнения их для подсчета значения расхождения, а также для создания тепловой карты для тестирования.

В начале работы класса инициализируется массив observers, затем он заполняется всеми объектами с компонентом Observer на текущей сцене. После этого все наблюдатели также добавляются в очередь observerQueue.

В каждом кадре работы программы система зрения проверяет, существуют ли в очереди ожидающие обработки наблюдатели. Если существуют и при этом значение activeGPURequests – «ложь», то из очереди удаляется первый наблюдатель, и, если он активен, система создает новую задачу из метода UpdateObserverAsync(), в ином случае наблюдатель возвращается в конец очереди.

Ключевым моментом оптимизации системы зрения является применение асинхронного программирования, которое позволяет выполнять части кода в побочных потоках программы, задействуя несколько ядер процессора, и тем самым не задерживать основной поток, значительно увеличивая как средний, так и минимальный показатели FPS. Чтобы добиться асинхронной работы системы зрения, метод UpdateObserverAsync(), а также многие методы, вызываемые в нем, были реализованы с помощью Awaitable – класса асинхронных задач в Unity, схожего с классом Task в языке программирования C#. Асинхронные классы возвращают объект класса Awaitable, а также задействуют ключевое слово await и методы Awaitable.MainThreadAsync() и Awaitable.BackgroundThreadAsync() для обеспечения асинхронной работы кода.

Метод UpdateObserverAsync() отвечает за обновление всех необходимых свойств наблюдателя. Первым шагом для этого является вызов у наблюдателя метода UpdateRenderTextures(). Поскольку метод выполняет запись отрисованных текстур в новый формат асинхронно, прежде чем продолжить их обработку, необходимо дождаться завершения возвращенных методом запросов, для этого используется метод WaitOnGPURequestsAsync() вместе с ключевым словом await. Как только выполнение запросов завершится, текстуры

TargetsPresentT2D и TargetsAbsentT2D сохранятся в переменные pTexture и aTexture соответственно.

Перед вызовом метода обновления текстур значение activeGPURequests приравниваем «истине», чтобы предотвратить обновление других наблюдателей в очереди и тем самым снизить максимальную нагрузку на видеокарту. После окончания выполнения асинхронных запросов к видеокарте значение activeGPURequests приравниваем «лжи».

Далее выполняем вызов асинхронного метода ProcessBlurAsync() для текстур pTexture и aTexture. Сначала в методе создаем представление текстуры в виде нативного массива colorArray из объектов класса Color32 путем вызова метода GetRawTextureData() у переданной текстуры. Поскольку данный метод может выполняться лишь в главном потоке программы, перед этим происходит вызов Awaitable.MainThreadAsync() с ключевым словом await.

Затем, если переданный радиус размытия больше нуля, сначала происходит переход в побочный поток программы при помощи кода await Awaitable.BackgroundThreadAsync(), после чего создается новый объект класса FastGaussianBlur на основании переданной текстуры, а затем возвращается результат вызова у класса размытия метода Process() с аргументом радиуса размытия, – все выполняется в асинхронном режиме. Таким образом, метод возвращает текстуру, обработанную классом FastGausianBlur, или же саму исходную текстуру в формате нативного массива из Color32, если переданный радиус размытия был неположительным.

После получения текстур в формате NativeArray<Color32> метод UpdateTexturesAsync() вызывает одну из перегрузок метода CalculateDiscrepancyAsync() в зависимости от того, необходимо ли создание тепловой карты. Обе перегрузки, в свою очередь, проводят синхронизацию с побочными потоками программы и совершают вызов соответствующей перегрузки метода CalculateDiscrepancy(), обе из которых выполняются по следующему алгоритму.

1. В качестве параметров принимают две текстуры в формате NativeArray<Color32>, которые и будут сравниваться между собой. Кроме того, принимается маска зрения, определяющая, какие пиксели будут игнорироваться, и формула расчета разницы пикселей.

2. Во вложенном цикле, проходящем по каждой координате пикселей двух текстур, происходит проверка значения соответствующей ячейки массива-маски. Если оно – «ложь», пиксель пропускается, иначе происходит расчет разности пикселей двух текстур по текущим координатам с применением переданной формулы. Полученная разность прибавляется к diff – накапливаемому значению расхождения.

3. Метод возвращает нормализованное значение расхождения, равное частному diff и количеству сравненных пикселей.

Различие двух перегрузок CalculateDiscrepancy() заключается в том, что одна из них принимает в виде аргумента ссылку на текстуру тепловой карты в виде нативного массиве из Color32 и в момент расчета значения каждого из пикселей записывает туда одно из двух значений: цвет HeatmapColdColor, если пиксель необходимо пропустить согласно маске или если разность пикселей меньше значения параметра NoiseFilter. В ином случае цвет пикселя рассчитывается при помощи линейной интерполяции двух цветов, взяв за начальный цвет параметр HeatmapColdColor, за конечный – HeatmapHotColor, а за величину интерполяции рассчитанное значение разности пикселей. На рис. 8 приведен пример тепловой карты, полученной с помощью интерполяции черного и белого цветов.

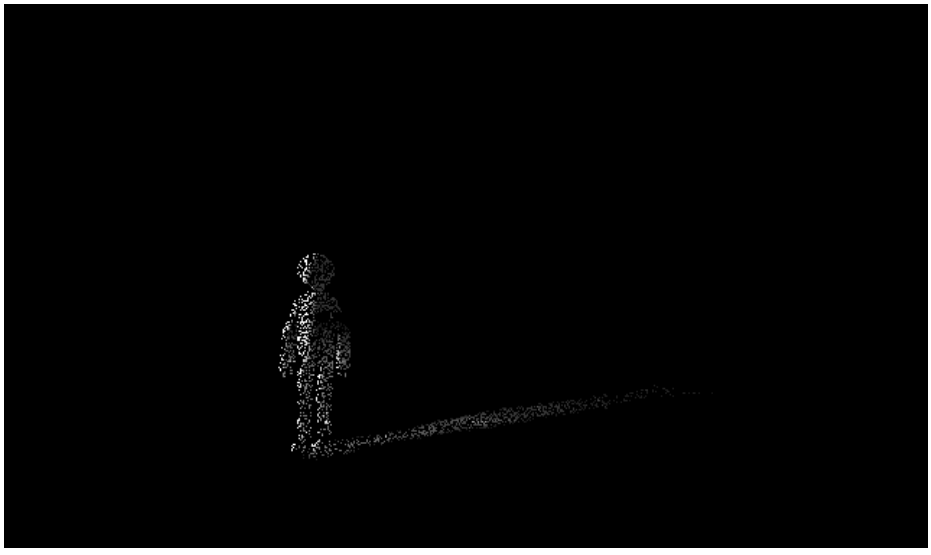

Рис. 8. Пример тепловой карты.

В текущей реализации системы зрения имеются два метода сравнения текстур, где R_1, G_1, B_1 – значения каналов RGB для i-го пикселя на 1-й текстуре,

R_2, G_2, B_2 – значения каналов RGB для i-го пикселя на 2-й текстуре, а V – разность двух пикселей:

- простая разность $V = \frac{|R_1-R_2|+|G_1-G_2|+|B_1-B_2|}{3 \cdot 255}$;
- среднеквадратичная разность $V = \frac{(R_1-R_2)^2+(G_1-G_2)^2+(B_1-B_2)^2}{3 \cdot 255}$.

После завершения расчета несоответствия текстур система, если этого требует параметр saveBlurredTextures, записывает в свойства наблюдателя текстуры, обработанные классом FastGaussianBlur, а также тепловую карту, если таковая была создана, все перечисленное может быть выполнено только при синхронизации с главным потоком программы, поэтому используем его лишь при тестировании системы.

Затем полученное значение несоответствия текстур сохраним у наблюдателя при помощи метода UpdateVisionDiscrepancy(), также происходит вызов события TexturesWereUpdated, и, наконец, наблюдатель возвращается в конец очереди observerQueue, завершая тем самым итерацию обновления наблюдателя.

ТЕСТИРОВАНИЕ

Подготовка тестирования

Первая тестовая сцена представляет собой изолированное помещение, специально разработанное для проверки работы алгоритмов обнаружения объектов. Пространство организовано таким образом, чтобы обеспечить контролируемые условия освещения и четко определенные зоны видимости, что крайне важно для точной оценки работы системы зрения.

Основой сцены служит закрытая комната, центральным элементом которой является разделительная стена, создающая четкую границу между освещенными и затемненными участками. В углу комнаты расположен объект Point Light с достаточной интенсивностью света для частичного освещения противоположного угла. Такая конфигурация позволяет моделировать различные сценарии обнаружения объектов как в хорошо освещенных зонах, так и в условиях недостаточной видимости (рис. 9).

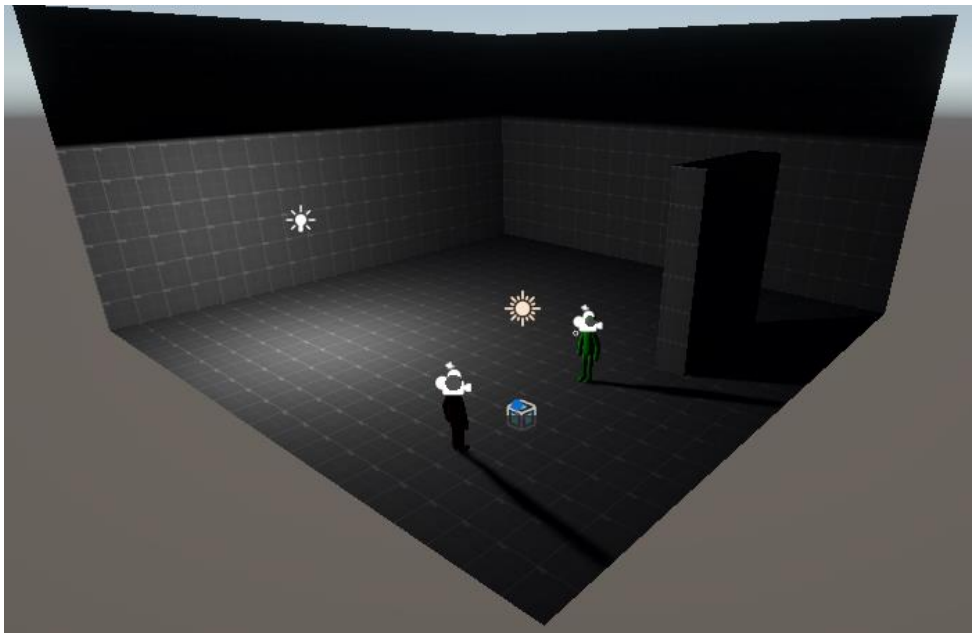

Рис. 9. Простая сцена для тестирования.

Для тестирования были использованы антропоморфные модели, состоящие из комбинации простых геометрических фигур. Модель игрока включает капсуловидное тело и сферическую голову с камерой, расположенной на уровне глаз. Управление персонажем реализовано через стандартный компонент CharacterController с дополнительным скриптом PlayerControl, обеспечивающим перемещение игрока и вращение камеры.

Особое внимание было уделено настройкам освещения сцены – интенсивность окружающего света специально была снижена до 0.1, а отражения полностью отключены, чтобы убедиться в том, что участки сцены, не освещенные ни одним источников света, будут предельно темными. Настройка Shadow – Depth Bias была установлена на значение 0.5, чтобы устранить такой артефакт рендеринга, как «теневой акне».

В качестве второй тестовой сцены, более сложной для отрисовки, был использован бесплатный ассет из Unity Asset Store со сценой, содержащей хижину, окруженную лесом. Эта сцена позволяет протестировать видимость при различных условиях, таких как разные уровни освещения, щели между досками хижины и густой туман (рис. 10).

Рис. 10. Сложная сцена для тестирования.

UIManager

Класс UIManager представляет собой центральный управляющий элемент пользовательского интерфейса для системы зрения NPC в Unity. Он тесно взаимодействует с VisionSystem и отдельными наблюдателями (Observer), предоставляя визуализацию и статистику в реальном времени.

Класс поддерживает две основные группы элементов интерфейса: текстовые статистические показатели и визуальные текстуры, дающие обработанное изображение из системы зрения.

В каждом кадре в методе Update() происходит несколько важных процессов. Система рассчитывает среднее значение FPS с помощью взвешенного скользящего среднего, где более свежие значения имеют больший вес. Если включен показ статистики наблюдателя, класс регулярно обновляет UI с заданным интервалом updateCooldown. Класс также реагирует на различные горячие клавиши для переключения визуализации текстур, отображения статистики, переключения тепловой карты и сохранения текущих текстур в файлы.

При изменении целевого наблюдателя через ChangeTargetedObserverer() метод перенастраивает все связанные элементы интерфейса, включая текстуры и их размеры, а также подписывается на события обновления текстур.

Метод UpdateObserverStatsUI() формирует подробную текстовую информацию о текущем состоянии наблюдателя. Это включает расчетное расхождение зрения, процент видимости, наличие прямой видимости, интервал обновления и уровень подозрения для каждой цели. Подозрение отображается в процентном соотношении к пороговому значению для тревоги.

Визуальная часть реализована в UpdateTextureUI(), который управляет отображением трех типов текстур: обычного изображения с целями, изображения без целей и тепловой карты. По запросу текстуры могут быть сохранены на диск в формате PNG.

Особенностью класса является разделение логики обновления UI на регулярные интервалы для статистики и мгновенные обновления для текстур по событию. Это позволило снизить нагрузку на процессор при сохранении актуальности визуальной информации.

Результаты тестирования

Тестирование было проведено в сборке приложения при следующих различных параметрах разных уровней нагрузки:

- один, шесть или восемнадцать наблюдателей;
- простая тестовая сцена или сложная;
- разрешение текстур отрисовки: 320 x 180 или 640 x 360;
- размытие включено или выключено для всех.

При всех комбинациях вышеперечисленных параметров были замерены среднее значение FPS, а также время реакции наблюдателей, т. е. время обработки одного наблюдателя. Полученные результаты были занесены в две различные таблицы: статистика FPS (табл. 1) и статистика времени реакции (табл. 2).

Табл. 1. Статистика среднего значения FPS.

Сцена	1 наблюдение, простая сцена	6 наблюдений, простая сцена	18 наблюдений, простая сцена	1 наблюдение, сложная сцена	6 наблюдений, сложная сцена	18 наблюдений, сложная сцена
320 x 180 без размытия	500	470	400	185	180	170
640 x 360 без размытия	500	500	450	190	175	165
320 x 180 с размытием	300	200	180	150	120	115
640 x 360 с размытием	180	85	80	100	55	55

Чтобы избежать влияния случайного шума, система зрения использовала значения Noise Filter, равное 0.1. Разные значения радиуса размытия наблюдателей не оказали влияния на производительность.

Табл. 2. Статистика среднего времени реакции (в миллисекундах).

Сцена	1 наблюдение, простая сцена	6 наблюдений, простая сцена	18 наблюдений, простая сцена	1 наблюдение, сложная сцена	6 наблюдений, сложная сцена	18 наблюдений, сложная сцена
320 x 180 без размытия	7	38	137	16	99	310
640 x 360 без размытия	12	45	150	21	103	317
320 x 180 с размытием	19	95	295	29	145	480
640 x 360 с размытием	50	200	700	57	300	850

На основе полученных данных можно сделать следующие выводы:

- увеличение числа наблюдателей умеренно уменьшает среднее значение FPS и линейно увеличивает время реакции каждого из них;

- сложность сцены оказывает сильное влияние на среднее значение FPS и умеренное – на время реакции наблюдателей;
- разрешение текстур отрисовки оказывает умеренное влияние на среднее значение FPS и время реакции без использования размытия и значимое влияние при использовании размытия;
- использование размытия оказывает значимое влияние как на среднее значение FPS, так и на среднее время реакции.

Оценка системы

Основной вывод, вытекающий из проведенных тестов, заключается в следующем: производительность системы предсказуема и зависит от контролируемых параметров, значит, ее легко оптимизировать для целей конкретного проекта.

В среднем система проигрывает стандартному подходу по значению FPS во время игры и по простоте настройки, однако это компенсируется точностью определения степени обнаружения, учетом сложных визуальных эффектов без необходимости программировать специальные условия, а также гибкостью существующих параметров.

Система показала себя наилучшим образом при небольшом числе наблюдателей, значит, идеально подходит для проектов или сцен, где одновременно присутствуют только один или несколько NPC, использующих систему зрения.

Поскольку сцену требуется отрисовывать до трех раз за один кадр, очень важно, чтобы каждая отрисовка происходила быстро, поэтому система лучше всего подходит для проектов с хорошо оптимизированной или простой, стилизованной графикой.

И, что наиболее важно, система помогает больше всего в тех проектах, где необходима точность и реалистичность визуального восприятия персонажей, а не простота. В первую очередь к таким проектам относятся реалистичные симуляторы и игры с глубокими механиками стелса.

ЗАКЛЮЧЕНИЕ

Представлен новый подход к реализации системы визуального восприятия для неигровых персонажей в видеоиграх. Разработанный алгоритм основан на отрисовке и сравнении двух изображений и учитывает сложные визуальные эффекты, такие как освещение, тени и маскировка, что значительно повышает реализм поведения NPC. Этот алгоритм обеспечивает более реалистичное поведение NPC по сравнению с традиционными методами, особенно в сложных условиях видимости, при этом сохраняя приемлемую производительность.

Предложенный подход может найти применение в различных жанрах видеоигр, особенно в играх с элементами скрытности, где реалистичное визуальное восприятие NPC является критическим фактором для создания увлекательного игрового процесса.

Перспективные пути развития системы включают создание меню отладки для удобства изменения и тестирования параметров в сборках приложения, применение модульного дизайна для упрощения интеграции системы в существующие проекты, использование параллелизации для большей части алгоритма, распределение нагрузки при проверках рейкастами, а также поиск новых способов оптимизации системы.

СПИСОК ЛИТЕРАТУРЫ

1. *Panwar H.* The NPC AI of The Last of Us: A case study // arXiv. 2022. arXiv:2207.00682. P. 1–7.

2. *Mahmoud I., Jaffal Y., Wloka D.* A Vision Simulation Algorithm for Non-Player Character in Static Scene // University of Kassel, Germany Kassel. 2014. P. 1–6.

3. *Vehkala M.* Creating the AI for the Living, Breathing World of Hitman: Absolution. GDC 2013. URL: https://www.gdcvault.com/play/1019353

4. *McIntosh T.* The Last of Us: Human Enemy AI. GDC 2014. URL: https://www.gdcvault.com/play/1020338/The-Last-of-Us-Human

5. *Walsh M.* Modeling AI Perception and Awareness in Splinter Cell: Blacklist. GDC 2014. URL: https://youtu.be/RFWrKHM0vAg

6. *McIntosh T.* Human Enemy AI in The Last of Us. In Game AI Pro 360. CRC Press. 2019. P. 13–24.

7. *Welsh, R.* Crytek's Target Tracks Perception System. In Game AI Pro: Collected Wisdom of Game AI Professionals, CRC Press. 2013. P. 403–411.

8. *Walsh M.* Modeling Perception and Awareness in Tom Clancy's Splinter Cell Blacklist // In Game AI Pro 360, CRC Press. 2019. P. 73–86.

9. *Ying Z., Edwards N., Kutuzov M.* Efficient Visibility Approximation for Game AI using Neural Omnidirectional Distance Fields // Proceedings of the ACM on Computer Graphics and Interactive Techniques. 2024. Vol. 7, No. 1. P. 1–15.

10. *Estgren M.* Modelling NPC perception using supervised learning. Uppsala University, Sweden Uppsala, 2021. 8 p.

11. *Bourg D.M., Seemann G.* AI for Game Developers. O'Reilly Media, Inc. 2004. 371 p.

12. *Примаченко А.М., Хафизов М.Р*. Разработка системы визуального восприятия игровых агентов в видеоиграх // Электронные библиотеки. 2025. Т. 28, № 3. С. 506–531.

13. NPC Eyes Sight System – PRO.
URL: https://www.fab.com/listings/6b54716a-dd21-414d-b78f-384068de14b7

14. *Erdelyi C.* Using Computer Vision Techniques to Play an Existing Video Game. California State University San Marcos, San Marcos, 2019. 49 p.

15. *Pramod R.T., Katti H., Arun S.P.* Human peripheral blur is optimal for object recognition // Vision Research. 2022. Vol. 200. No. 108083.

16. *Jarosz W.* Fast Image Convolutions // ACM SIGGRAPH University of Illinois Urbana-Champaign. 2001. 11 p.

17. Fastest Gaussian Blur (in Linear Time).
URL: https://blog.ivank.net/fastest-gaussian-blur.html

EVALUATION OF THE PROSPECTIVITY OF A HYBRID MODEL MODELING OF VISUAL PERCEPTION BY GAME AGENTS USING TEXTURE DRAWING

A. M. Primachenko[0009-0008-6396-2035]

Kazan (Volga region) Federal University, Kazan, Russia

primachenko.artem@mail.ru

Abstract

This article presents a newly developed visual perception algorithm for game agents, implemented in the Unity game engine. The proposed method is based on comparing images from two cameras, taking into account complex visual effects (lighting, shadows, occlusion), and is supplemented by line-of-sight checks, consideration of object velocity, and gradual detection mechanics. The algorithm was optimized using asynchronous computations, dynamic camera activation, and accelerated algorithms. The system was tested under various load levels, and conclusions were drawn regarding the optimal conditions for the algorithm's operation. The paper also analyzes the scientific literature on similar solutions, identifying their strengths and weaknesses. The results can be applied in video game development to create realistic behavior for non-player characters, especially in games with stealth elements.

Keywords*: video games, artificial intelligence, perception system, NPC, Unity, computer vision, game design.*

REFERENCES

1. *Panwar H.* The NPC AI of The Last of Us: A case study // arXiv. 2022. arXiv:2207.00682. P. 1–7.

2. *Mahmoud I., Jaffal Y., Wloka D.* A Vision Simulation Algorithm for Non-Player Character in Static Scene // University of Kassel, Germany Kassel. 2014. P. 1–6.

3. *Vehkala M.* Creating the AI for the Living, Breathing World of Hitman: Absolution. GDC 2013. URL: https://www.gdcvault.com/play/1019353

4. *McIntosh T.* The Last of Us: Human Enemy AI. GDC 2014.
URL: https://www.gdcvault.com/play/1020338/The-Last-of-Us-Human

5. *Walsh M.* Modeling AI Perception and Awareness in Splinter Cell: Blacklist. GDC 2014. URL: https://youtu.be/RFWrKHM0vAg

6. *McIntosh T.* Human Enemy AI in The Last of Us. In Game AI Pro 360. CRC Press. 2019. P. 13–24.

7. *Welsh, R.* Crytek's Target Tracks Perception System. In Game AI Pro: Collected Wisdom of Game AI Professionals, CRC Press. 2013. P. 403–411.

8. *Walsh M.* Modeling Perception and Awareness in Tom Clancy's Splinter Cell Blacklist // In Game AI Pro 360, CRC Press. 2019. P. 73–86.

9. *Ying Z., Edwards N., Kutuzov M.* Efficient Visibility Approximation for Game AI using Neural Omnidirectional Distance Fields // Proceedings of the ACM on Computer Graphics and Interactive Techniques. 2024. Vol. 7, No. 1. P. 1–15.

10. *Estgren M.* Modelling NPC perception using supervised learning. Uppsala University, Sweden Uppsala, 2021. 8 p.

11. *Bourg D.M., Seemann G.* AI for Game Developers. O'Reilly Media, Inc. 2004. 371 p.

12. *Primachenko A.M., Hafizov M.R.* Development of a Visual Perception System for Game Agents in Video Games // Russian Digital Libraries Journal. 2025. Vol. 28, № 3. P. 506–531.

13. NPC Eyes Sight System – PRO.
URL: https://www.fab.com/listings/6b54716a-dd21-414d-b78f-384068de14b7

14. *Erdelyi C.* Using Computer Vision Techniques to Play an Existing Video Game. California State University San Marcos, San Marcos, 2019. 49 p.

15. *Pramod R.T., Katti H., Arun S.P.* Human peripheral blur is optimal for object recognition // Vision Research. 2022. Vol. 200. No. 108083.

16. *Jarosz W.* Fast Image Convolutions // ACM SIGGRAPH University of Illinois Urbana-Champaign. 2001. 11 p.

17. Fastest Gaussian Blur (in Linear Time).
URL: https://blog.ivank.net/fastest-gaussian-blur.html

СВЕДЕНИЯ ОБ АВТОРЕ

ПРИМАЧЕНКО Артем Михайлович – аспирант Института информационных технологий и интеллектуальных систем Казанского федерального университета. Направление исследований: визуальное восприятие игровых агентов в видеоиграх.

Artyom Mikhailovich PRIMACHENKO – PhD student at the Institute of Information Technologies and Intelligent Systems, Kazan Federal University. Research area: visual perception of game agents in video games.

email: primachenko.artem@mail.ru

ORCID: 0009-0008-6396-2035

Материал поступил в редакцию 24 апреля 2026 года