# Квартальный план разработки RTS-проекта (Unity 2D) для начинающего

Цель плана: За три месяца поэтапно обучить джуниор-разработчика созданию архитектурно сложного 2D RTS-проекта на Unity. План разбит на модули по ключевым подсистемам игры (цикл симуляции, экономика, инфекция, события, ИИ, пути и оптимизация). Каждый модуль содержит описание, учебные цели, задачи с возрастающей сложностью, пошаговые указания, советы по логированию и отладке, рекомендации по использованию AI-помощника (Codex GPT), а также способы тестирования. Модульная структура позволит постепенно наращивать навыки – от базовых механик до продвинутых оптимизаций – с обязательным пониманием и проверкой результатов на каждом этапе.

Следуя этому техническому плану, новичок сможет уверенно двигаться от создания каркаса проекта к полноценному участию в разработке сложной RTS. Важно: Не переходить к следующему модулю, пока текущий полностью не освоен и отлажен.

## Модуль 1: Базовый цикл симуляции и управление временем

### Краткое описание модуля

В этом модуле реализуется основа игрового цикла – фиксированный цикл симуляции, независимый от частоты кадров, с возможностью паузы и изменения скорости времени. Мы создадим собственный SimulationLoop – компонент Unity, который с заданной частотой (например, 10 тиков в секунду) обновляет игровую логику, даже если FPS непостоянен[[1]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=%D0%A6%D0%B5%D0%BB%D1%8C%3A%20%D1%81%D0%B8%D0%BC%D1%83%D0%BB%D1%8F%D1%86%D0%B8%D1%8F%20%D0%B8%D0%B4%D1%91%D1%82%20%D1%84%D0%B8%D0%BA%D1%81%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D1%8B%D0%BC%20%D1%88%D0%B0%D0%B3%D0%BE%D0%BC,%D0%BE%D1%81%D1%82%D0%B0%D0%BD%D0%B0%D0%B2%D0%BB%D0%B8%D0%B2%D0%B0%D0%B5%D1%82%20%D1%81%D0%B8%D0%BC%D1%83%D0%BB%D1%8F%D1%86%D0%B8%D1%8E%2C%20%D0%BD%D0%BE%20%D0%BD%D0%B5%20UI). Пауза должна останавливать симуляцию, не затрагивая работу интерфейса (UI)[[1]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=%D0%A6%D0%B5%D0%BB%D1%8C%3A%20%D1%81%D0%B8%D0%BC%D1%83%D0%BB%D1%8F%D1%86%D0%B8%D1%8F%20%D0%B8%D0%B4%D1%91%D1%82%20%D1%84%D0%B8%D0%BA%D1%81%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D1%8B%D0%BC%20%D1%88%D0%B0%D0%B3%D0%BE%D0%BC,%D0%BE%D1%81%D1%82%D0%B0%D0%BD%D0%B0%D0%B2%D0%BB%D0%B8%D0%B2%D0%B0%D0%B5%D1%82%20%D1%81%D0%B8%D0%BC%D1%83%D0%BB%D1%8F%D1%86%D0%B8%D1%8E%2C%20%D0%BD%D0%BE%20%D0%BD%D0%B5%20UI). Также добавим функции изменения скорости (замедление/ускорение) и «пошагового» режима (шаг по одному тику).

Этот фундаментальный шаг научит контролировать время в игре и закладывает каркас для всех последующих систем (экономики, ИИ и т.д.), которые будут вызываться в цикле.

### Цели обучения

* Понять концепцию фиксированного тика симуляции (fixed update loop) и зачем он нужен для RTS.
* Научиться останавливать и запускать симуляцию отдельно от рендеринга (пауза без паузы UI).
* Реализовать гибкое управление скоростью игры (разные коэффициенты времени).
* Получить навыки базовой работы с Unity MonoBehaviour (метод Update()), таймерами (Time.unscaledDeltaTime) и Debug-инструментами.
* Научиться проверять корректность игрового цикла при разных условиях (низкий/высокий FPS, пауза, одиночный шаг).

### Основные задачи (по возрастанию сложности)

1. Создание проекта и сцены. Инициализируйте Unity-проект и настройте структуру папок согласно плану архитектуры. Создайте пустую сцену с базовым окружением:
2. Настройте Tilemap для игрового поля (пока просто пустая сетка)[[2]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=,%E2%80%94%20DOTS%2FEntities%20%D0%B4%D0%BB%D1%8F%20%D1%84%D0%B8%D0%BA%D1%81%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B3%D0%BE%20%D1%82%D0%B8%D0%BA%D0%B0)[[3]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=%D0%94%D0%B5%D0%BD%D1%8C%201%E2%80%932%20,%D1%82%D0%B8%D0%BA%D0%BE%D0%BC%20%D0%B8%20%D0%BF%D0%B0%D1%83%D0%B7%D0%BE%D0%B9). Убедитесь, что единицы измерения (например, 1 тайл = 1 метр) понятны – это пригодится для перемещения объектов.
3. Создайте в сцене объект SimulationLoop с компонентом MonoBehaviour. Этот объект будет глобальным «таймером» симуляции.
4. Разработайте интерфейс IClock (например, с свойствами Delta и IsPaused) для абстракции источника тиков[[4]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=%60%60%60csharp%20public%20interface%20IClock%20,IsPaused%20%3F%200f%20%3A%20tick). Реализация этого интерфейса в SimulationLoop поможет передавать дельту времени другим системам.
5. Реализация фиксированного цикла. В скрипте SimulationLoop реализуйте логику накопления времени и вызова тиков:
6. Добавьте поле tick (длительность шага симуляции, например 0.1 сек для 10 тиков/сек) и аккумулятор acc[[5]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=public%20sealed%20class%20SimulationLoop%20%3A,IsPaused%20%3F%200f%20%3A%20tick).
7. В Update() каждый кадр накапливайте невзвешенное время: acc += Time.unscaledDeltaTime. Используем unscaledDeltaTime, чтобы пауза через timeScale не влияла на наш цикл[[6]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=,%D0%BB%D1%83%D1%87%D1%88%D0%B5%20%D0%BA%D0%BE%D0%BD%D1%82%D1%80%D0%BE%D0%BB%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D1%82%D1%8C%20%D1%81%D0%B8%D0%BC%D1%83%D0%BB%D1%8F%D1%86%D0%B8%D1%8E%20%D1%8F%D0%B2%D0%BD%D1%8B%D0%BC%20%D1%86%D0%B8%D0%BA%D0%BB%D0%BE%D0%BC).
8. Пока накопилось времени больше одного шага (acc >= tick), запускайте циклы симуляции по одному тику:
   * Вычитайте из аккумулятора величину tick и затем вызывайте обновление всех систем игры в определённом порядке. Пока системы (экономика, инфекция, ИИ и др.) не реализованы, можете временно вызывать заглушки или логировать вызов тикov для каждой подсистемы. Позже, по мере прохождения модулей, сюда будут добавляться реальные вызовы: InfectionSystem.Tick(tick), EconomySystem.Tick(tick) и т.д.[[7]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=while%20%28acc%20,Flush%28%29%3B). Сейчас достаточно вывести в лог, например, "Tick X processed", где X – номер тика.
   * После обновления систем в конце каждого тика очищайте очередь событий: вызов EventBus.Flush(). (EventBus будет реализован в отдельном модуле, но сразу предусмотрите его вызов для правильного порядка событий[[7]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=while%20%28acc%20,Flush%28%29%3B)).
9. Реализуйте возможность паузы: флаг IsPaused. В начале Update() проверяйте: если игра на паузе, не накапливайте время и сразу возвращайтесь (пропуская логику тика)[[8]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=void%20Update%28%29%20,Tick%28tick). В интерфейсе IClock сделайте Delta = IsPaused ? 0 : tick для удобства (или просто используйте флаг).
10. Добавьте метод SetPaused(bool) для изменения состояния паузы[[9]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=TacticalAI.Tick%28tick%29%3B%20EventBus.Flush%28%29%3B%20,IsPaused%20%3D%20v). Его будем вызывать из UI или горячей клавиши.
11. Управление скоростью и одиночный шаг. Расширьте SimulationLoop для поддержки разных скоростей и пошагового режима:
12. Скорости ×0.5, ×1, ×2, ×4: Введите множитель времени (например, timeMultiplier), который будет влиять на скорость симуляции. Можно просто делить/умножать значение tick или накапливаемое время. Например, для ускорения ×2 уменьшайте tick вдвое (или увеличивайте accumulator в 2 раза быстрее). Учтите, что UI и анимации должны продолжать работать с нормальной скоростью (они и так на unscaled время, так что изменение нашего tick не затронет UI)[[10]](file://file_00000000e2c461f4bb2cbf5057f4f2ac#:~:text=2,%D1%86%D0%B8%D0%BA%D0%BB%D1%83).
13. Одиночный тик (Step): Реализуйте отдельную функцию или кнопку, по нажатию которой симуляция выполняет ровно один тик и снова останавливается. Для этого можно: если игра на паузе, принудительно вызвать один цикл while(acc >= tick) либо вызвать методы Tick() всех систем один раз вручную. Сделайте проверку (Assert), что при одном нажатии выполняется не более одного тика[[11]](file://file_0000000069fc624685d94f067e42a757#:~:text=3%29%20%D0%9F%D0%B0%D1%83%D0%B7%D0%B0%3A%20%D0%BF%D1%80%D0%BE%D0%B2%D0%B5%D1%80%D1%8C%2C%20%D1%87%D1%82%D0%BE%20,%EE%88%80cite%EE%88%82turn0search12%EE%88%81).
14. Добавьте UI-контролы или горячие клавиши для управления скоростью и паузой. Например, кнопки «Пауза/Продолжить», «Шаг», а также переключатели скорости. Эти элементы можно разместить в простом Overlay-меню для разработчика.
15. Логирование и визуальные индикаторы (для отладки цикла):
16. Счётчик тиков: Добавьте на экран простой текстовый индикатор (например, в углу UI) или Debug.Log вывод, который увеличивается каждый раз, когда происходит тик[[12]](file://file_0000000069fc624685d94f067e42a757#:~:text=,tick). Это поможет отследить, что тики идут регулярно и не пропускаются. Можно хранить счётчик tickCount в SimulationLoop и обновлять его в цикле.
17. Лог порядка систем: В начале и конце каждого тика выводите в лог сообщения, например: "Tick #N start", а после вызова всех систем – "Tick #N end (flushed events X)". Это подтвердит, что порядок вызовов устойчивый и EventBus flush выполняется в конце тика, а не в середине.
18. Индикатор паузы: отображайте на экране статус (например, надпись "PAUSED"), когда игра на паузе, чтобы визуально убеждаться, что пауза работает.
19. Проверка стабильности тика: Путём резкого изменения FPS (например, включить/выключить VSync, переключить настройки качества) проверьте, что частота тиков остаётся около заявленных 10 тиков/сек без дрожания. Счётчик тиков и время можно сравнить: за 5 секунд реального времени должно накопиться ~50 тиков при ×1 скорости.
20. Профилирование производительности цикла (базовое измерение):
21. Откройте Unity Profiler (Window → Analysis → Profiler, либо клавиша Ctrl+7) и запустите игру на несколько секунд[[13]](file://file_0000000069fc624685d94f067e42a757#:~:text=6%29%20%D0%9F%D1%80%D0%BE%D1%84%D0%B8%D0%BB%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5%3A%20,%EE%88%80cite%EE%88%82turn0search12%EE%88%81). Посмотрите график времени на вкладках Scripts и UI. Должно быть консистентное время выполнения логики каждого тика и отсутствие скачков загрузки сборщика мусора (GC)[[13]](file://file_0000000069fc624685d94f067e42a757#:~:text=6%29%20%D0%9F%D1%80%D0%BE%D1%84%D0%B8%D0%BB%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5%3A%20,%EE%88%80cite%EE%88%82turn0search12%EE%88%81). На таком раннем этапе нагрузка минимальна, но полезно привыкнуть анализировать кадры.
22. Удостоверьтесь, что GC не генерируется каждый тик из-за ваших операций. Например, если логируете каждую итерацию while, это может создавать сборки мусора из-за конкатенации строк. В продакшн-коде постоянный лог внутри цикла следует отключить или обернуть условием if (Debug.isDebugBuild).

### Логирование и отладка

* Индикатор тиков: Проверьте, что UI-индикатор или лог счётчика тиков увеличивается равномерно и без пропусков. Если при паузе он останавливается, а при изменении скорости – меняет темп, всё правильно.
* Пауза: Во время паузы убедитесь, что метод Update() сразу выходит (ни одного тика не совершается). При этом анимации UI (если есть) должны продолжать проигрываться. Если UI-анимации замирают на паузе, убедитесь, что у Animator и таймеров использованы unscaled time (например, Animator.updateMode = UnscaledTime, WaitForSecondsRealtime вместо обычного)[[14]](file://file_0000000069fc624685d94f067e42a757#:~:text=,%D0%A3%D0%B1%D0%B5%D0%B4%D0%B8%D1%81%D1%8C%2C%20%D1%87%D1%82%D0%BE%20%D0%B8%D0%B3%D1%80%D0%BE%D0%B2%D0%BE%D0%B9%20%D0%BA%D0%BE%D0%B4).
* Одиночный шаг: Нажатие кнопки «Step» должно приводить к выполнению ровно одного цикла while(acc>=tick). После шага цикл не должен сразу продолжаться, иначе это признак, что неправильно контролируется пауза или accumulator. Проверьте через Assert, что за один вызов Step производится не более 1 тика[[11]](file://file_0000000069fc624685d94f067e42a757#:~:text=3%29%20%D0%9F%D0%B0%D1%83%D0%B7%D0%B0%3A%20%D0%BF%D1%80%D0%BE%D0%B2%D0%B5%D1%80%D1%8C%2C%20%D1%87%D1%82%D0%BE%20,%EE%88%80cite%EE%88%82turn0search12%EE%88%81).
* Стабильность при разных FPS: Сравните поведение тиков при 30 FPS и при 200 FPS (можно ограничить FPS через QualitySettings или vsync). Число тиков в секунду и общее развитие симуляции должно быть одинаковым. Это ключевое свойство фиксированного цикла – независимость от частоты кадров[[1]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=%D0%A6%D0%B5%D0%BB%D1%8C%3A%20%D1%81%D0%B8%D0%BC%D1%83%D0%BB%D1%8F%D1%86%D0%B8%D1%8F%20%D0%B8%D0%B4%D1%91%D1%82%20%D1%84%D0%B8%D0%BA%D1%81%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D1%8B%D0%BC%20%D1%88%D0%B0%D0%B3%D0%BE%D0%BC,%D0%BE%D1%81%D1%82%D0%B0%D0%BD%D0%B0%D0%B2%D0%BB%D0%B8%D0%B2%D0%B0%D0%B5%D1%82%20%D1%81%D0%B8%D0%BC%D1%83%D0%BB%D1%8F%D1%86%D0%B8%D1%8E%2C%20%D0%BD%D0%BE%20%D0%BD%D0%B5%20UI).
* Типичные ошибки: Если замечаете, что тики «плавают» или идут быстрее/медленнее заявленного – проверьте сброс аккумулятора в цикле while. Нередко ошибка – не полностью вычитать tick из acc, что приводит к ускорению или замедлению цикла[[15]](file://file_0000000069fc624685d94f067e42a757#:~:text=%EE%88%80cite%EE%88%82turn0search12%EE%88%81). Другой частый баг – забыть вызвать EventBus.Flush() в конце тика (пока EventBus не работает, это не влияет, но позже будет критично)[[16]](file://file_0000000069fc624685d94f067e42a757#:~:text=,%D0%B2%20%D0%BA%D0%BE%D0%BD%D1%86%D0%B5%20%D1%82%D0%B8%D0%BA%D0%B0). Также убедитесь, что симуляционное время отделено от времени UI: не используйте Time.deltaTime там, где нужен unscaledDeltaTime[[6]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=,%D0%BB%D1%83%D1%87%D1%88%D0%B5%20%D0%BA%D0%BE%D0%BD%D1%82%D1%80%D0%BE%D0%BB%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D1%82%D1%8C%20%D1%81%D0%B8%D0%BC%D1%83%D0%BB%D1%8F%D1%86%D0%B8%D1%8E%20%D1%8F%D0%B2%D0%BD%D1%8B%D0%BC%20%D1%86%D0%B8%D0%BA%D0%BB%D0%BE%D0%BC).

### Применение Codex GPT

* Генерация кода цикла: Вы можете воспользоваться AI-помощником, чтобы получить шаблон класса. Например, спросите: «Напиши C# класс MonoBehaviour SimulationLoop с фиксированным шагом 0.1с и паузой». Codex предложит реализацию (сверьте с планом и примером из документа)[[4]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=%60%60%60csharp%20public%20interface%20IClock%20,IsPaused%20%3F%200f%20%3A%20tick). Полученный код нужно внимательно проверить и протестировать.
* Исправление ошибок: Если цикл работает не так, как ожидалось (например, ticks пропускаются или идут неритмично), скопируйте свой код и запросите: «Найди ошибку в логике фиксированного цикла». Расширение Codex может подсветить проблемные места (например, неправильный сброс acc или использование deltaTime вместо unscaledDeltaTime).
* Пояснение кода: Если какие-то конструкции непонятны (например, Time.unscaledDeltaTime или работа while цикла), можно спросить: «Объясни, как работает этот код строчка за строчкой». Это поможет глубже понять шаблон фиксированного тика.
* Создание UI через Codex: Не уверены, как добавить UI-кнопки или текстовый индикатор? Спросите: «Как в Unity через код создать на экране текст для отображения счётчика тиков?». Codex может подсказать использование Canvas и TextMeshPro, либо изменение текста через скрипт.
* Сниппеты для проверки: Также Codex поможет со скриптами отладки – например, команду для печати текущего Application.targetFrameRate или для переключения VSync через скрипт, что пригодится при тестировании.

### Тестирование и проверка результата

* Функциональный тест цикла: Запустите игру и вручную проверьте сценарии:
* Без паузы, нормальная скорость: тик-счётчик растёт примерно на 10 в секунду (если tick=0.1). Через ~5 секунд должно быть ~50 тиков.
* Пауза: при включении паузы счётчик останавливается, при снятии – продолжает расти.
* Одиночный шаг: при паузе нажатие «Step» увеличивает счётчик ровно на 1 и не больше.
* Ускорение ×2: тиковый счётчик начинает расти примерно в 2 раза быстрее (≈20 тиков/сек). Замедление ×0.5 – в 2 раза медленнее (≈5 тиков/сек).
* Тест с переменным FPS: Включите Stats в Game View (показывает текущий FPS). Изменяйте нагрузку (например, сожмите окно игры для увеличения FPS или включите интенсивную графику для просадки FPS) и наблюдайте за счётчиком тиков. Он должен быть стабильным. Например, если на высокой нагрузке FPS упал до 30, тики всё равно должны идти 10 раз/сек, просто по несколько за кадр. Если при просадке FPS игра «замедляется», значит, цикл не успевает нагнать – нужно проверить логику accumulator.
* Profiler: Запись профайлера должна показать регулярные всплески нагрузки в скриптах каждые 0.1 сек. В разделе Timeline профайлера можно увидеть, что за кадр может выполняться несколько тиков (несколько вызовов методов Tick подряд) – это нормальное поведение при фиксированном цикле, когда FPS ниже частоты тиков. Убедитесь, что время на 10 тиков ~равномерно распределено, а не накапливается бесконечно.
* Unity Frame Debugger: На этом этапе рендеринг минимален, но вы можете открыть Frame Debugger и убедиться, что Tilemap отрисовывается оптимально (Tilemap можно оставить режим Chunk для минимизации draw calls[[17]](file://file_00000000e2c461f4bb2cbf5057f4f2ac#:~:text=%EE%88%80cite%EE%88%82turn6search6%EE%88%82turn6search0%EE%88%81.%20,%D0%B2%D1%8B%D0%B8%D0%B3%D1%80%D1%8B%D1%88%20%D0%BD%D0%B0%20%D0%B2%D0%B0%D1%88%D0%B5%D0%B9%20%D1%81%D1%86%D0%B5%D0%BD%D0%B5%20%EE%88%80cite%EE%88%82turn3search13%EE%88%82turn3search0%EE%88%82turn3search1%EE%88%81), если уже есть несколько слоёв). Это скорее ознакомительно – что профилировщик и дебаггер кадров работают, и вы умеете их открывать.
* Перед переходом дальше: Добейтесь, чтобы механизм времени работал полностью предсказуемо. Критерии готовности: пауза без багов, изменение скорости влияет только на симуляцию, в профайлере нет бесконтрольного роста нагрузки или утечек, а вы уверены в том, как устроен ваш цикл. Только после этого переходите к введению игровых данных и систем.

## Модуль 2: Данные и конфигурации (ScriptableObject → Runtime-модели)

### Краткое описание модуля

Второй модуль посвящён управлению данными игры: создание конфигураций с помощью Unity ScriptableObjects и использование их для инициализации чистых C#-моделей (POCO) на старте игры. Мы подготовим "каталоги" игровых объектов (юнитов, зданий, технологий, параметров инфекции и т.п.) в виде ScriptableObject-ассетов, которые легко редактировать в Unity Editor. При запуске эти ассеты будут скопированы в обычные объекты (runtime-модели), с которыми и будет работать симуляция[[18]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=3%29%20%D0%94%D0%B0%D0%BD%D0%BD%D1%8B%D0%B5%20%D0%B8%20%D0%BA%D0%BE%D0%BD%D1%84%D0%B8%D0%B3%D0%B8%20,%D1%81%D1%82%D0%B0%D1%80%D1%82%D0%B5%20%D0%BF%D1%80%D0%B5%D0%B2%D1%80%D0%B0%D1%89%D0%B0%D0%B5%D0%BC%20%D0%B2%20%D1%87%D0%B8%D1%81%D1%82%D1%8B%D0%B5%20%D0%BE%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D1%8B)[[19]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=%2F%2F%20Domain%2FTech%2FTech,string%3E%20Prereq%29%3B). Такой подход предотвращает нежелательную мутацию ассетов во время игры и упрощает сохранение состояния[[20]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=%D0%9F%D0%BE%D1%87%D0%B5%D0%BC%D1%83%20%D1%82%D0%B0%D0%BA%3A%20SO%20%E2%80%94%20%D1%83%D0%B4%D0%BE%D0%B1%D0%B5%D0%BD,%D0%BA%D0%BE%D0%BF%D0%B8%D0%B8%20%E2%80%94%20%D0%B1%D0%B5%D0%B7%D0%BE%D0%BF%D0%B0%D1%81%D0%BD%D1%8B%20%D0%B8%20%D1%81%D0%B5%D1%80%D0%B8%D0%B0%D0%BB%D0%B8%D0%B7%D1%83%D0%B5%D0%BC%D1%8B).

Кроме того, в этом модуле заложим основу системы сохранения (Save/Load) – продумаем формат сохранения игрового состояния в JSON и версионирование данных. Пока игра простая, можно реализовать базовое сохранение экономики и параметров инфекции, чтобы убедиться в работоспособности подхода.

### Цели обучения

* Научиться создавать ScriptableObject-ассеты и скрипты для игровых конфигураций.
* Понимать разницу между данными на этапе дизайна (ассеты) и данными в рантайме (копии, которые можно изменять).
* Реализовать загрузку конфигураций при старте игры: из ассетов в структуры, используемые системами (экономика, инфекция, технологии и т.д.).
* Понять ограничения Unity JsonUtility при сериализации и подготовить структуры данных с учётом этих ограничений (например, без Dictionary).
* Ввести концепцию версии сохранения и научиться делать миграцию при изменении формата данных.
* Закрепить навык написания простых юнит-тестов для проверки корректности моделей (например, сериализация/десериализация).

### Основные задачи (по возрастанию сложности)

1. Определение структур данных. Выделите основные типы игровых данных, которые будут конфигурироваться через редактор:
2. Примеры: параметры инфекции (базовая скорость заражения, радиус распространения и т.п.), характеристики юнитов и зданий (стоимость, прочность), технологии (названия, стоимость исследования, требования).
3. Для каждого типа данных создайте класс ScriptableObject. Например, InfectionParams (ScriptableObject) с полями для коэффициентов модели заражения, или BuildingDef с полями стоимости строительства, радиуса действия и пр. В документе архитектуры есть пример TechDef – ScriptableObject для технологии[[21]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=%2F%2F%20Data%2FTechs%2FTechDef.asset%20%28SO%29%20%5BCreateAssetMenu%28menuName%3D,Prereq%3B).
4. Пометьте класс атрибутом [CreateAssetMenu] для удобного создания ассетов из меню Unity[[21]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=%2F%2F%20Data%2FTechs%2FTechDef.asset%20%28SO%29%20%5BCreateAssetMenu%28menuName%3D,Prereq%3B). Добавьте необходимые поля (Id, название, числовые параметры, ссылки на связанные объекты).
5. Runtime-модель: для каждого ScriptableObject определите соответствующую чистую C# модель (можно struct или record). Например, для TechDef – класс или record Tech с аналогичными полями[[19]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=%2F%2F%20Domain%2FTech%2FTech,string%3E%20Prereq%29%3B), но вместо ссылок на другие SO храните, например, списки ID или индексы (т.к. runtime-объекты будут связываться между собой иначе).
6. Связи и идентификаторы: продумайте систему идентификаторов. Например, у каждой технологии TechDef.Id – уникальный строковый ID[[21]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=%2F%2F%20Data%2FTechs%2FTechDef.asset%20%28SO%29%20%5BCreateAssetMenu%28menuName%3D,Prereq%3B), то же для юнитов/зданий. Эти ID можно использовать как ключи при преобразовании в runtime-модели (например, список prerequisite-технологий хранить как список строк ID или enum, вместо прямых ссылок на SO, чтобы легко сериализовать).
7. Создание ассетов и загрузка. В Unity Editor создайте несколько экземпляров ваших ScriptableObject для теста:
8. Например, сделайте 2-3 технологии, парочку зданий и задайте им поля.
9. Создайте менеджер загрузки, который в методе Start() находит все необходимые ассеты и создает из них runtime-копии. Способы загрузки:
   * Простой вариант: поместите ассеты в Resources папку и вызывайте Resources.LoadAll<T>() для каждого типа (например, всех TechDef)[[22]](file://file_00000000e2c461f4bb2cbf5057f4f2ac#:~:text=%2A%2A%D0%A7%D1%82%D0%BE%20%D0%B4%D0%BE%D0%B1%D0%B0%D0%B2%D0%B8%D1%82%D1%8C%20%28%D0%BC%D0%B5%D1%85%D0%B0%D0%BD%D0%B8%D0%BA%D0%B8%29%3A%2A%2A%20,%D0%BC%D0%B8%D0%B3%D1%80%D0%B0%D1%86%D0%B8%D0%B8%20%D0%BF%D1%80%D0%B8%20%D0%B7%D0%B0%D0%B3%D1%80%D1%83%D0%B7%D0%BA%D0%B5).
   * Более продвинутый: заведите ScriptableObject "каталог" – например, GameConfig с полями-списками (List<TechDef> allTechs, etc.), и вручную заполните в инспекторе ссылки. В Start просто переберите эти списки.
10. При загрузке каждой записи:
    * Создайте соответствующий runtime-объект (например, new Tech(id, cost, prereqList) для TechDef[[19]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=%2F%2F%20Domain%2FTech%2FTech,string%3E%20Prereq%29%3B)).
    * Сохраняйте их в удобной структуре, например, словарь Dictionary<string, Tech> для технологий по ID, чтобы потом быстро получать по идентификатору. Учтите: словари нельзя напрямую сериализовать через JsonUtility, но для быстрого доступа в памяти они удобны. Позже при сохранении нужно будет перенести их в список или другой формат.
    * Не используйте сами ScriptableObject напрямую в системе – всегда работайте с копиями. Добавьте проверку (например, в режиме редактора if(Application.isPlaying) Assert.IsFalse(AssetDatabase.Contains(obj))) что объект, с которым вы работаете, не является ассетом[[23]](file://file_0000000069fc624685d94f067e42a757#:~:text=1%29%20,%D0%BA%D0%BE%D1%80%D1%80%D0%B5%D0%BA%D1%82%D0%BD%D0%BE%20%D0%B2%D0%BE%D1%81%D1%81%D1%82%D0%B0%D0%BD%D0%B0%D0%B2%D0%BB%D0%B8%D0%B2%D0%B0%D0%B5%D1%82%20%D0%BF%D0%BE%D0%B4%D1%82%D0%B8%D0%BF%D1%8B). Это гарантирует: вы не мутируете оригинальные конфиги во время игры.
11. Глобальное хранилище состояния. Спроектируйте класс GameState или несколько классов, описывающих текущее состояние игры, которое нужно будет сохранять:
12. Например, GameState может содержать: текущее время/тик, состояния всех клеток (сетка инфекции), состояния экономики (балансы бюджетов), выполненные исследования, построенные здания и т.д.
13. На этом этапе у нас реализованы не все системы, поэтому начните с того, что есть:
    * Состояние инфекции: можно хранить как двумерный массив или список значений заражения по каждой клетке. Например, infectionGrid – массив float или небольшой класс CellState (уровень заражения + статус S/I/R).
    * Состояние экономики: достаточно сохранить текущие балансы двух экономик.
    * Можно добавить список выполненных технологий (пока у нас нет механики исследований, но на будущее).
    * Версия сейва: добавьте поле SaveVersion (например, int) в GameState. При будущих изменениях структуры данных вы будете проверять версию и делать миграцию при загрузке[[24]](file://file_0000000069fc624685d94f067e42a757#:~:text=%D0%B0%D0%B4%D0%B0%D0%BF%D1%82%D0%B5%D1%80%2C%20%D0%BB%D0%B8%D0%B1%D0%BE%20%D1%81%D1%82%D0%BE%D1%80%D0%BE%D0%BD%D0%BD%D0%B8%D0%B9%20JSON,%D0%B7%D0%B0%D0%B3%D1%80%D1%83%D0%B7%D0%BA%D0%B5%20%E2%80%94%20%D0%BC%D0%B8%D0%B3%D1%80%D0%B0%D1%86%D0%B8%D1%8F%20%D0%BF%D0%BE%20%D1%81%D0%B2%D0%B8%D1%82%D1%87%D1%83).
14. Сделайте GameState [Serializable], чтобы JsonUtility могла его сериализовать. Если в нём есть сложные поля (например, Dictionary), замените или пометьте их как [NonSerialized] и заполняйте через вспомогательные списки. Важно: JsonUtility не поддерживает Dictionary и полиморфные типы без специальных атрибутов[[25]](file://file_0000000069fc624685d94f067e42a757#:~:text=%21AssetDatabase.Contains%28obj%29%29,%D0%BA%D0%BE%D1%80%D1%80%D0%B5%D0%BA%D1%82%D0%BD%D0%BE%20%D0%B2%D0%BE%D1%81%D1%81%D1%82%D0%B0%D0%BD%D0%B0%D0%B2%D0%BB%D0%B8%D0%B2%D0%B0%D0%B5%D1%82%20%D0%BF%D0%BE%D0%B4%D1%82%D0%B8%D0%BF%D1%8B.%20%EE%88%80cite%EE%88%82turn0search15%EE%88%81). Поэтому убедитесь, что все поля GameState – либо примитивы/структуры, либо списки/массивы таких примитивов/структур.
15. Реализация сохранения и загрузки. Добавьте в проект скрипт SaveLoadService (или методы в существующем менеджере), реализующие:
16. Сохранение: сбор текущего состояния игры в экземпляр GameState и сериализация его в JSON. Используйте JsonUtility.ToJson(gameState). Полученную строку можно сохранить в файл (например, в Application.persistentDataPath) или просто в PlayerPrefs для простоты.
17. Загрузка: чтение JSON из файла, десериализация JsonUtility.FromJson<GameState>(json). После этого нужно применить загруженное состояние к текущей игре:
    * Обновить все системы: перезаполнить сетку инфекции значениями из save, установить балансы экономик, отметить выполненные технологии и т.д. Проще всего это сделать, уничтожив текущую сцену и создав новую, инициализировав всё заново на основе GameState (особенно если структура сложная). Но можно и на лету применять.
18. Миграция версий: если GameState.SaveVersion загруженного файла меньше текущей версии, выполните необходимые преобразования (например, заполнение новых полей значениями по умолчанию). Пока можно просто логировать предупреждение, что версия не совпадает, так как формат ещё не менялся. Но заложить эту логику важно на будущее[[24]](file://file_0000000069fc624685d94f067e42a757#:~:text=%D0%B0%D0%B4%D0%B0%D0%BF%D1%82%D0%B5%D1%80%2C%20%D0%BB%D0%B8%D0%B1%D0%BE%20%D1%81%D1%82%D0%BE%D1%80%D0%BE%D0%BD%D0%BD%D0%B8%D0%B9%20JSON,%D0%B7%D0%B0%D0%B3%D1%80%D1%83%D0%B7%D0%BA%D0%B5%20%E2%80%94%20%D0%BC%D0%B8%D0%B3%D1%80%D0%B0%D1%86%D0%B8%D1%8F%20%D0%BF%D0%BE%20%D1%81%D0%B2%D0%B8%D1%82%D1%87%D1%83).
19. Безопасная точка сохранения: Решите, как игрок инициирует сохранение. Хороший подход – сохранять только когда игра остановлена или на паузе, чтобы не захватить промежуточное состояние. Либо дождаться конца тика (после EventBus.Flush()) и затем снять снимок состояния[[26]](file://file_0000000069fc624685d94f067e42a757#:~:text=%60,%EE%88%80cite%EE%88%82turn1search6%EE%88%81). Сейчас, для упрощения, можно сохранять по кнопке, которая ставит игру на паузу и вызывает Save.
20. Дополнительная задача (опционально): интегрируйте Addressables для загрузки ассетов, если планируете много контента:
21. Это продвинутый шаг: вместо Resources использовать Addressables, что потребует инициализации Addressables API, но даст больше контроля над памятью.
22. Если уже знакомы: реализуйте загрузку конфигов через Addressables.LoadAssetsAsync<T> и не забудьте правильно выгружать их через Release, иначе может быть утечка памяти[[27]](file://file_00000000e2c461f4bb2cbf5057f4f2ac#:~:text=,%D0%BF%D0%BE%20%D1%81%D0%BF%D0%BE%D1%81%D0%BE%D0%B1%D0%B0%D0%BC%20%D0%B7%D0%B0%D0%B3%D1%80%D1%83%D0%B7%D0%BA%D0%B8%2F%D1%87%D0%B0%D1%81%D1%82%D0%BE%D1%82%D0%B5%20%D0%B8%D1%81%D0%BF%D0%BE%D0%BB%D1%8C%D0%B7%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F%20%EE%88%80cite%EE%88%82turn2search10%EE%88%82turn2search16%EE%88%81)[[28]](file://file_0000000069fc624685d94f067e42a757#:~:text=,%EE%88%80cite%EE%88%82turn1search10%EE%88%81).
23. Добавьте счётчики или логи сколько ассетов загружено/выгружено, используя events или отладочный UI, чтобы убедиться, что нет избыточно висящих в памяти объектов[[28]](file://file_0000000069fc624685d94f067e42a757#:~:text=,%EE%88%80cite%EE%88%82turn1search10%EE%88%81).

### Пошаговые указания к реализации

* Создание ScriptableObject: В Unity выберите Create → C# Script, назовите его, например, TechDef.cs. Наследуйте класс от ScriptableObject. Добавьте [CreateAssetMenu(menuName = "Defs/Tech")] сверху, чтобы в меню Assets → Create появилось создание Tech. Опишите поля: строка Id, название, числовые параметры, ссылки на другие ScriptableObject (например, список prereq типа TechDef[])[[21]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=%2F%2F%20Data%2FTechs%2FTechDef.asset%20%28SO%29%20%5BCreateAssetMenu%28menuName%3D,Prereq%3B). Скомпилируйте, затем правой кнопкой в папке создайте пару таких ассетов и заполните поля в инспекторе.
* Runtime-модель: В папке Scripts/Domain/ соответствующей создайте класс или struct Tech (лучше struct или record, т.к. данные иммутабельны). Дайте ему такие же поля (Id, cost, prerequisites), но в prerequisites храните список строк ID или других простых значений[[19]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=%2F%2F%20Domain%2FTech%2FTech,string%3E%20Prereq%29%3B). Почему не держать ссылки на объекты? – чтобы этот класс был “чистым” и не включал UnityEngine.ScriptableObject (что затруднит сериализацию).
* Загрузка данных: Создайте скрипт GameDataLoader (MonoBehaviour на отдельном объекте менеджере или внутри SimulationLoop). В его Awake или Start получите списки ассетов. Если используете Resources, то: TechDef[] techs = Resources.LoadAll<TechDef>(""); (путь можно уточнить). Далее инициализируйте, например, Dictionary<string, Tech> techMap = new(); и цикл: foreach(var techDef in techs) { techMap[techDef.Id] = new Tech(techDef.Id, techDef.ResearchCost, techDef.Prereq.Select(t => t.Id).ToList()); }. Аналогично для других типов (Buildings, InfectionParams и т.п.). Теперь ваши системы могут обращаться к этим структурам для получения параметров.
* Использование данных в системах: Подумайте, как передать эти конфиги туда, где они нужны. Например, InfectionSystem может хранить ссылку на загруженный InfectionParams runtime-объект, EconomySystem – на настройки экономики (если такие есть). Можно сделать так, что GameDataLoader после загрузки устанавливает нужные поля в синглтонах систем или через событие сообщает о готовности данных.
* Сохранение GameState: Реализовав минимум игровых систем (после модулей 3 и 4 у нас будет экономика и инфекция), вернитесь к задаче сохранения. Например, после модуля 4, когда будет что сохранять, заполните класс GameState:
* tickCount или текущее время симуляции,
* infectionGrid (возможно, сериализуем как массив float или int статусов; большие двумерные массивы JsonUtility тоже умеет),
* stateEconomyBalance и playerEconomyBalance,
* возможно, список построек или технологий (если к тому времени появятся, иначе пропустите).
* JSON тест: Напишите небольшой метод отладки: при нажатии некой клавиши сериализуйте GameState и сразу десериализуйте обратно, сравните некоторые поля. Например, посчитайте сумму заражения по grid до сохранения и после загрузки – должна совпадать. Это выявит, если какая-то часть не сохраняется. Также проверьте, что поля как Id и списки правильно сохраняются (в JSON они должны присутствовать).
* Версионирование: Установите SaveVersion = 1 в текущем GameState. Сохраните игру, измените версию на 2 (например, добавив новое поле) – попробуйте загрузить старый JSON. Вы должны детектировать, что SaveVersion=1 (старый) и обработать: например, заполнить новое поле дефолтным значением, затем установить версию к объекту 2. Этот опыт с ручной миграцией пригодится, когда реальные изменения будут необходимы.

### Логирование и отладка

* Мутация ассетов: В отладочном режиме (только в редакторе) проверьте, что нигде в коде вы не изменяете ScriptableObject. Помочь может Assert, предложенный в документе[[23]](file://file_0000000069fc624685d94f067e42a757#:~:text=1%29%20,%D0%BA%D0%BE%D1%80%D1%80%D0%B5%D0%BA%D1%82%D0%BD%D0%BE%20%D0%B2%D0%BE%D1%81%D1%81%D1%82%D0%B0%D0%BD%D0%B0%D0%B2%D0%BB%D0%B8%D0%B2%D0%B0%D0%B5%D1%82%20%D0%BF%D0%BE%D0%B4%D1%82%D0%B8%D0%BF%D1%8B): после создания runtime-копии можно проверить через UnityEditor.AssetDatabase.Contains(obj) – вернёт true, если объект является ассетом. В рантайме все наши рабочие объекты должны не быть ассетами. Если где-то осталась ссылка на оригинальный SO и вы её модифицируете, это ошибка.
* JsonUtility ограничения: Попробуйте прямо сейчас выполнить JsonUtility.ToJson(someScriptableObject) – убедитесь, что, например, поля-словарики там не сохраняются. Если ваши runtime-модели содержат несериализуемые поля, подумайте, как их исключить или преобразовать. Например, Dictionary заменить на список пар ключ-значение при сохранении/загрузке. Лучше выявить и решить эти моменты заранее, написав простые тесты на сериализацию.
* Тест сериализации: Напишите EditMode юнит-тест: создайте программно объект GameState, заполните его несколькими значениями (например, занесите 2 значения в infectionGrid, балансы экономик). Преобразуйте в JSON и обратно, убедитесь, что полученные объекты равны исходным по данным. Особенно если используете [SerializeReference] для полиморфизма – проверьте, что JsonUtility восстанавливает подтипы корректно[[25]](file://file_0000000069fc624685d94f067e42a757#:~:text=%21AssetDatabase.Contains%28obj%29%29,%D0%BA%D0%BE%D1%80%D1%80%D0%B5%D0%BA%D1%82%D0%BD%D0%BE%20%D0%B2%D0%BE%D1%81%D1%81%D1%82%D0%B0%D0%BD%D0%B0%D0%B2%D0%BB%D0%B8%D0%B2%D0%B0%D0%B5%D1%82%20%D0%BF%D0%BE%D0%B4%D1%82%D0%B8%D0%BF%D1%8B.%20%EE%88%80cite%EE%88%82turn0search15%EE%88%81).
* Загрузка данных: Отладьте загрузку конфигов – выведите в лог количество загруженных тех же технологий, зданий и пр., сравните с числом ассетов в проекте. Все должны загрузиться.
* Integrity check: Если у TechDef были ссылки на prereq, убедитесь, что в runtime Tech эти prereq действительно сохранены (например, размер списков совпадает). При сложных связях (например, технология A требует B и C) – убедитесь, что ваши runtime-модели могут это выразить (возможно, хранением ID B и C).
* Частые проблемы: забыли пометить класс [Serializable] – тогда JsonUtility выдаст пустой JSON. Или, например, попытались сериализовать что-то вроде Dictionary – JSON выйдет пустой. Лечится заменой структур. Другая ошибка – сериализация больших структур (сетка 1000x1000) может быть медленной или громоздкой; но пока игра маленькая, это приемлемо.
* Адресация ассетов: Если применили Addressables, включите режим PlayMode (Use Existing Build или Simulate) и отследите, что после сцены, где вы загрузили ассеты, при выходе или переходе они выгружаются. Логи или профайлер памяти помогут заметить, не остались ли ассеты висеть (у Addressables есть профайл и EventViewer, но это за рамками базового плана).

### Применение Codex GPT

* Создание классов: Используйте Codex, чтобы ускорить шаблонный код. Например, запрос: «Создай класс ScriptableObject InfectionParams с полями float baseInfectionRate, float spreadRadius». Помощник сгенерирует простой класс, вам останется только скорректировать названия/тип данных. Аналогично можно сгенерировать класс runtime-модели: «Напиши C# record InfectionSettings с такими же полями».
* Сбор загрузки: Если не уверены, как программно загрузить ассеты, спросите: «Как загрузить все ScriptableObject определённого типа в Unity?». Codex может подсказать про Resources.LoadAll или адресаблы.
* JSON сериализация: Попросите пример: «Пример использования JsonUtility для сохранения и загрузки класса GameState». Получив код, убедитесь, что он учитывает ToJson/FromJson. Codex может сразу показать, как записать файл через File.WriteAllText и прочитать через File.ReadAllText – можете скопировать и адаптировать под пути.
* Unit-тесты: Пусть Codex поможет написать юнит-тест: «Напиши NUnit-тест, который проверяет, что JsonUtility корректно сериализует и десериализует объект с полями X, Y». Результат поправьте под свои классы. Это избавит от необходимости вспоминать синтаксис Assertions.
* Поиск ошибок сериализации: Если при сохранении что-то не сохраняется, вы можете показать Codex JSON вывод и код GameState, спросив: «Почему поле X не появляется в JSON?». Часто он подскажет, что, к примеру, свойство не сериализуется (сериализуются только поля) или тип не поддерживается.
* Оптимизация данных: Codex способен объяснить и рекомендуемые практики. Например: «Почему Unity не сохраняет Dictionary в JSON и как это обойти?» – поможет понять и выбрать решение (конверсия или использование другой JSON библиотеки).
* Работа с Addressables: Можно даже спросить: «Пример кода загрузки ScriptableObject через Addressables и выгрузки» – AI-ассистент подскажет нужные методы (Addressables.LoadAssetAsync, Addressables.Release).

### Тестирование и проверка результата

* Проверка загрузки конфигов: Запустите сцену и в консоли проверьте логи: должно выводиться, сколько объектов загружено и какие ID у них. Сравните с вашими созданными ассетами – все ли считаны. Если чего-то не хватает, убедитесь, что ассеты находятся в Resources или привязаны к сцене/ScriptableObject-реестру.
* Тест GameState Save/Load: Когда будет реализована базовая экономика и инфекция (после модулей 3 и 4), протестируйте полный цикл:
* Запустите игру, немного измените состояние (например, повысится заражение в нескольких клетках, изменятся балансы после каких-то трат).
* Нажмите кнопку Save (или вызовите метод через консоль/дебаг).
* Сбросьте состояние – например, перезапустите сцену или вызовите метод Load сразу после Save (имитируя загрузку).
* Сравните: совпадают ли ключевые показатели? Число заражённых клеток, значения балансов и т.п. должны быть такими же, как в момент сохранения.
* Юнит-тесты: Запустите написанные EditMode тесты. Они должны пройти, подтверждая, что (де)сериализация данных работает.
* Проверка данных в инспекторе: На время отладки можно сделать так, чтобы при загрузке GameState игра ставилась на паузу. Тогда можно в Unity Editor открыть объекты (например, InfectionSystem) и убедиться, что их поля соответствуют загруженному состоянию. Это ручной способ верификации, но наглядный.
* Перед переходом дальше: Убедитесь, что вы уверенно получаете нужные параметры из конфигов внутри кода – например, InfectionSystem может обратиться к infectionParams.baseInfectionRate и получить значение, которое вы задавали в ассете. Сохранение на этом этапе может быть не полностью реализовано (если, скажем, ещё нечего сохранять), но каркас должен быть готов. Критерии готовности: все ключевые игровые параметры берутся не из «магических чисел» в коде, а из конфигурационных объектов; изменение значения в ассете меняет поведение в игре. Сохраняемый GameState по крайней мере включает основные части (таймер, экономику, инфекцию) и может быть успешно выгружен/загружен без ошибок.

## Модуль 3: Экономическая система (двойное ВВП)

### Краткое описание модуля

В этом модуле реализуем экономику игры, причём сразу в двух аспектах: экономика государства (ИИ) и экономика игрока. Концепция «двойного ВВП» предполагает, что у государства и игрока отдельные бюджеты и модели принятия решений[[29]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=,%D1%83%D0%BA%D1%80%D0%B5%D0%BF%D0%BB%D0%B5%D0%BD%D0%B8%D0%B9%20%D0%B8%20%C2%AB%D1%82%D0%BE%D1%87%D0%B5%D1%87%D0%BD%D1%83%D1%8E%20%D0%BF%D0%BE%D0%B4%D0%B4%D0%B5%D1%80%D0%B6%D0%BA%D1%83%C2%BB%20%D0%98%D0%98). Государственный ИИ собирает ресурсы (налоги, производство) и тратит их на стратегические нужды – оборону, карантины, исследования базового уровня. Игрок же распоряжается личным бюджетом, вкладывая его в критические исследования, постройки и поддержку, тем самым влияя точечно. Эти две экономики взаимодействуют (например, государство может субсидировать игрока, игрок – платить налоги, или штрафы за провалы).

Мы начнём с упрощённой модели: две независимых экономики, каждая со своим балансом и базовыми операциями – резервирование средств и окончательное списание при оплате действий. Затем можно наращивать сложность: добавить пассивный доход для гос-экономики, механизмы займов для игрока, контракты с KPI (если проект позволяет). Экономика интегрируется с другими системами: строительство объектов должно тратить деньги, ИИ – выбирать, на что потратить бюджет, и т.д.

### Цели обучения

* Понять принципы построения финансовой системы в игре: учёт ресурсов, транзакции, инварианты (сохранение баланса).
* Создать интерфейс и реализацию экономики (IEconomy) с основными операциями (Balance, TryReserve, Commit, Tick)[[30]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=%60%60%60csharp%20public%20interface%20IEconomy%20,float%20dt%29%3B).
* Научиться обрабатывать ошибки транзакций (например, нехватка средств) и логировать их.
* Освоить написание модульных тестов для бизнес-логики (проверка инвариантов, корректности резервирования).
* Понять, как несколько экономических субъектов могут взаимодействовать, и подготовить почву для принятия решений ИИ на основе бюджетов.
* Интегрировать экономику в цикл симуляции (ежетиковые обновления, например начисление ресурсов).
* Освоить использование Codex для генерации шаблонного кода интерфейсов и тестов.

### Основные задачи (по возрастанию сложности)

1. Интерфейс экономики. Определите интерфейс IEconomy с необходимыми методами[[30]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=%60%60%60csharp%20public%20interface%20IEconomy%20,float%20dt%29%3B):
2. Свойство Balance (текущее количество ресурсов/денег).
3. Метод bool TryReserve(float amount): попытка зарезервировать сумму. Логика: если на счету достаточно средств, зарезервировать их под будущую операцию и вернуть true; иначе вернуть false. На этом этапе можно реализовать резервирование просто как проверку баланса (и, например, временное уменьшение баланса или отдельный счетчик резерва).
4. Метод void Commit(float amount): окончательно списывает ранее зарезервированные средства. Предполагается, что этот метод вызывается после успешного TryReserve, чтобы подтвердить трату денег.
5. Метод void Tick(float dt): обновление экономики каждый тик. Пока можно оставить пустым или использовать для пассивного дохода/расхода.
6. Создайте класс Economy реализующий IEconomy. В конструкторе или инициализации задавайте стартовый баланс. Реализуйте хранилища: например, храните balance и reserved суммы. На упрощённом этапе можно при TryReserve просто уменьшать balance сразу (имитируя мгновенное резервирование), а Commit не менять баланс (т.к. уже вычтено). Либо не уменьшать при резерве, а уменьшить при Commit – но тогда надо хранить сколько зарезервировано. Выберите подход и задокументируйте его, главное – после Commit общий баланс должен уменьшиться на сумму, а если TryReserve не был выполнен – баланс не меняется.
7. Добавьте удобство: метод AddFunds(float amount) для пополнения (например, налоговые поступления), чтобы модифицировать баланс извне.
8. Две реализации или два экземпляра. Решите, как отличить экономику государства и игрока:
9. Более гибко: сделайте два класса, наследующие от IEconomy, например StateEconomy и PlayerEconomy, которые могут переопределять поведение Tick (например, в StateEconomy Tick начисляет доход от налогов, а в PlayerEconomy – нет либо начисляет проценты по кредитам). На первое время можно не усложнять и использовать один класс Economy для обоих, просто с разными настройками.
10. Создайте в игре два объекта/сервиса: stateEconomy и playerEconomy. Их начальные балансы можно задать в конфиге (например, ScriptableObject с начальными финансами) или в коде (какими-нибудь значениями для теста).
11. Убедитесь, что эти объекты доступны остальным системам: например, BuildService (будет позже) сможет получить ссылку на тот IEconomy, который оплачивает строительство (игрок или государство).
12. Интеграция в SimulationLoop. Подключите экономику к циклу симуляции:
13. Добавьте вызов stateEconomy.Tick(dt) и playerEconomy.Tick(dt) в метод SimulationLoop.Update, внутри цикла тиков[[31]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=%2F%2F%20%D0%B2%D1%8B%D0%B7%D1%8B%D0%B2%D0%B0%D0%B5%D0%BC%20%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%BD%D1%8B%D0%B5%20%D1%82%D0%B8%D0%BA%D0%B8%20%D0%BF%D0%BE,Flush%28%29%3B). На ранней стадии Tick может быть пустым, но позже вы добавите в него логику.
14. Решите, нужна ли экономике более редкая частота обновления. В RTS часто экономика не обновляется каждый кадр: можно, например, прибавлять доход раз в 10 тиков. Если хотите, можете реализовать счетчик внутри Economy.Tick: суммировать dt и, скажем, раз в 1 секунду (если dt суммарно ≥1) добавлять доход. Либо просто каждый тик маленькую сумму (эквивалентно).
15. Простейший пример: пусть StateEconomy каждую секунду получает +10 единиц (налоги), а PlayerEconomy не получает пассивного дохода. Тогда в Tick можно написать: накопить время и когда >=1, увеличить баланс.
16. Убедитесь, что SimulationLoop передаёт dt правильно (будет фиксированное tick из вашего SimulationLoop, например 0.1).
17. Логирование транзакций. Добавьте в методы TryReserve/Commit вывод в лог для важных случаев:
18. При неудачном TryReserve логируйте предупреждение: "[Economy] ReserveFail: amount=..., balance=..., who=Player/State"[[32]](file://file_0000000069fc624685d94f067e42a757#:~:text=1%29%20Unit,%C2%AB%D0%B4%D0%B5%D0%BD%D1%8C%D0%B3%D0%B8%20%D0%BD%D0%B5%20%D0%B8%D1%81%D1%87%D0%B5%D0%B7%D0%B0%D1%8E%D1%82%2F%D0%BD%D0%B5%20%D0%BF%D0%BE%D1%8F%D0%B2%D0%BB%D1%8F%D1%8E%D1%82%D1%81%D1%8F%C2%BB). Это поможет отлаживать ситуацию, когда что-то не строится из-за нехватки денег.
19. Можно логировать и успешные транзакции для трассировки (но не перестарайтесь, чтобы лог не захлебнулся – возможно, только в режиме дебага).
20. Если реализуете резервирование с задержкой: логируйте и Commit: например, "Commit: ... funds deducted".
21. Если будут сложные кейсы (например, штрафы или возвраты), тоже логируйте.
22. Unit-тесты инвариантов. Напишите тесты, чтобы убедиться в корректности работы экономики:
23. Тест: после TryReserve (успешного) баланс не должен уменьшиться (если вы выбрали так) или уменьшился ровно на сумму (если списываете сразу) – зависит от вашей логики. Главное – после соответствующего Commit суммарное снижение равно сумме транзакции. Например: старт 100, TryReserve(30) -> balance либо 70 (если сразу списали), либо 100 (если пока держите на резервах), затем Commit(30) -> баланс 70 (во втором случае). В любом случае, деньги не должны исчезать или создаваться из ниоткуда – сумма списанная должна равняться сумме транзакции.
24. Тест: если два раза зарезервировать по 50 при балансе 100, второй раз должен провалиться (если не было Commit между ними). Баланс после первого резерва (если вычитаете) будет 50, второй TryReserve(50) -> true, но третий TryReserve(10) должен уже вернуть false.
25. Тест: Commit без предшествующего TryReserve – либо не вызывается, либо игнорируется. Можно написать, что Commit не должен делать отрицательный баланс, даже если вызвать напрямую.
26. Дымовой тест: симулируйте 100 тиков, в каждом случайным образом тратьте что-то (если хватает средств). К концу проверьте, что баланс уменьшился на сумму всех успешных расходов минус доходы, и что ни один баланс не ушёл в минус. Это поможет выявить логические ошибки (например, двойное списание).
27. Расширение механик (необязательно, продвинутые задачи):
28. Приоритеты гос-бюджета: В будущем, когда будет ИИ, можно сделать распределение бюджета по статьям. Сейчас можно заложить структуру: например, StateEconomy хранит доли бюджета на оборону, карантин, исследования. Пока не используем, но архитектурно предусмотрите (например, поля-проценты).
29. Займы и проценты: Для PlayerEconomy добавьте возможность уходить в отрицательный баланс (кредит) до определённого лимита, с тем чтобы каждую секунду начислялся процент и уменьшал баланс еще больше, если долг. Это сложнее, но если есть интерес – реализуйте в Tick логику: если balance < 0, balance -= balanceinterestRatedt (т.е. вычитание процента, который увеличивает долг).
30. Контракты с KPI: Это совсем продвинуто – завести сущность "Контракт" (например, государство заключает контракт с игроком построить защиту в регионе за 100 ед., если игрок не справился – штраф). Если интересно, пропишите класс Contract с полями цель, награда, штраф, дедлайн. Но реализацию можно отложить до появления ИИ, сейчас просто учитывайте, что бюджетные механики могут включать штрафы и бонусы.

### Пошаговые указания к реализации

* Создание IEconomy: В папке Scripts/Domain/Economy создайте интерфейс IEconomy и опишите сигнатуры методов[[30]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=%60%60%60csharp%20public%20interface%20IEconomy%20,float%20dt%29%3B). Затем в Scripts/Application или где уместно – класс Economy : IEconomy.
* Реализация Economy: Решите схему с резервами:
* Вариант A (простой): при TryReserve(amount) сразу уменьшать balance на amount, если хватает, и возвращать true; если не хватает – возвращать false, ничего не меняя. А Commit(amount) в таком случае может вообще ничего не делать (или просто для симметрии, хотя можно оставить пустым). Этот вариант проще, но терминология "Reserve/Commit" теряет смысл – вы сразу тратите деньги на этапе резерва. Однако для начала так можно.
* Вариант B (двухфазный): TryReserve только проверяет и, допустим, добавляет в некоторую переменную reserved += amount (или ставит флаг, что столько-то зарезервировано), но не списывает. Commit уже уменьшает balance на фактически зарезервированную сумму. Тут надо быть осторожным: а если за время между TryReserve и Commit баланс изменился? Но в однопоточной игре между ними обычно ничего не вставляется, кроме, возможно, другого TryReserve. Можно не углубляться: предположим, что TryReserve и Commit вызываются подряд без других операций.
* Реализуйте один из вариантов. Прокомментируйте в коде, как оно работает.
* Два экземпляра: Создайте два объекта Economy:
* Либо через добавление двух полей в существующий MonoBehaviour GameManager, которые в инспекторе можно заполнить (но лучше создать их программно).
* Например, в GameDataLoader после загрузки конфигов: stateEconomy = new Economy(initialStateFunds); playerEconomy = new Economy(initialPlayerFunds);. Или напишите классы StateEconomy/PlayerEconomy, если хотите переопределить Tick: stateEconomy.Tick начисляет, скажем, +1 каждый тик (или +10 каждую секунду как выше).
* Подключите эти объекты к SimulationLoop: либо передайте SimulationLoop ссылки, либо сделайте их синглтонами. Самый простой путь: сделайте их статическими полями для начала (но это хуже для тестов). Либо храните в SimulationLoop ссылки на обе экономики, и заполняйте их при старте.
* Начисление дохода: В методе Tick у StateEconomy добавьте: incomeAccumulator += dt; if(incomeAccumulator >= 1) { incomeAccumulator -= 1; balance += incomePerSec; }. Так каждую игровую секунду (не зависимо от FPS) будет приходить фиксированная сумма. У игрока можно ничего не делать или аналогично (если есть, например, пассивный доход).
* Подготовка к тестам: Убедитесь, что можно создать Economy и вызывать ее методы вне Unity (сделайте конструктор без зависимостей UnityEngine). Тогда вы легко протестируете логику в EditMode тестах.
* Юнит-тесты: В папке Assets/Tests (или через Window → General → Test Runner) создайте EditMode тесты. Например, EconomyTests:
* Тест ReserveDoesNotChangeBalance (если вы выбрали вариант B): создайте экономику с балансом 100, вызовите TryReserve(30) – ожидайте, что возвращено true, а balance всё ещё 100 (если вы не списываете сразу). Commit(30) – после него balance 70. И второй кейс: если TryReserve(200) – должно вернуть false и balance остаться 100.
* Тест MultipleReservesAndCommits: например, TryReserve(50) x2. Если вы списываете сразу, то первый вернёт true и баланс 50, второй – true и баланс 0 (если позволили ровно занулить). Если не списываете, можно имитировать контракт: TryReserve(50) -> true, TryReserve(60) -> false (потому что зарезервировано уже 50 и больше нельзя).
* Тест NoNegativeBalance: манипуляциями убедитесь, что после любых операций Balance >= 0 (если конечно вы не допускаете кредиты).
* Логирование: Внутри Economy добавьте Debug.LogWarning или Log для отказов. Чтобы в лог было понятно, какая экономика, добавьте либо разный префикс (например, сделать свойство Name или передавать в конструктор "Player" vs "State"). Лог на успехи можно делать через Log с уровнем Developer (например, Debug.Log("[Economy:Player] -50 for building X")).
* Связь с другими системами: Продумайте наперёд: экономика должна взаимодействовать с системой строительства (Module 5) и ИИ (Module 6). Это значит, что там будет вызываться if(playerEconomy.TryReserve(cost)) ... Commit(cost). Чтобы это сделать, те системы должны знать о playerEconomy. Вы можете внедрить зависимости, например, передать Economy ссылку в конструктор BuildService или сделать BuildService методом внутри Economy (нежелательно). Пока можно оставить глобальные ссылки или синглтон GameManager, откуда статически доступ к экономике.

### Логирование и отладка

* Unit-тесты: Запустите созданные тесты в Test Runner. Если что-то падает – исправьте логику. Например, частая проблема – забыли учесть граничные условия (ноль на счету, ровно равная сумма и т.п.).
* Интеграционный тест: Напишите короткий сценарий: пусть каждый тик государство получает +1, а игрок пытается тратить 0.2. Запустите 100 тиков и посмотрите на балансы:
* У государства должно прибавиться примерно +100\*0.1 (если 10 тик/сек) \* 1 ед./сек = +10 (зависит от реализации).
* У игрока если каждый тик тратит 0.2, то за 100 тиков потратит 20, если хватило. Проверьте, не ушёл ли он в минус. Если ушёл, значит дозволяете, либо ошибка в проверке.
* Логи отказов: Принудительно создайте ситуацию, где расходов больше, чем денег. Например, дайте игроку 10, а попытайтесь потратить 15. Должно в консоли появиться предупреждение [Economy] ReserveFail: amount=15, balance=10, who=Player[[33]](file://file_0000000069fc624685d94f067e42a757#:~:text=%E2%86%92%20%D0%B8%D1%82%D0%BE%D0%B3%D0%BE%D0%B2%D1%8B%D0%B5%20%D0%B1%D0%B0%D0%BB%D0%B0%D0%BD%D1%81%D1%8B%20%D0%BF%D1%80%D0%B5%D0%B4%D1%81%D0%BA%D0%B0%D0%B7%D1%83%D0%B5%D0%BC%D1%8B,%D0%BF%D1%80%D0%BE%D0%B2%D0%B5%D1%80%D1%8C%2C%20%D1%87%D1%82%D0%BE%20%D0%BD%D0%B5%D1%82%20%D0%B4%D1%80%D0%BE%D0%B6%D0%B0%D0%BD%D0%B8%D1%8F%20%D0%BF%D1%80%D0%B8%D0%BE%D1%80%D0%B8%D1%82%D0%B5%D1%82%D0%BE%D0%B2). Убедитесь, что сообщение информативно.
* Стресс-тест баланса: Если у государства много дохода и много расходов, не возникает ли багов? Попробуйте резко увеличить доход (например, 100 ед./сек) и одновременно тратить 50/сек, посмотрите в профайлер – не проседает ли (экономика код лёгкий, но вдруг). Заодно убедитесь, что нигде не появляется бесконечного роста памяти.
* Частые проблемы:
* Накопление ошибок округления: если dt=0.1, за 10 тиков incomeAccumulator = 1.0, все ок. Но если dt не точно 0.1 (что может быть, если accumulator работает с флотами), за N тиков может накопиться чуть больше 1.0 и дать лишний доход. Это не критично, но осознайте: такая погрешность возможна. В финансовой системе иногда используют целые числа (копейки, центы) или Decimal. В игре можно закрыть глаза или периодически нормализовать accumulator.
* Несоответствие Reserve/Commit: например, вызвали TryReserve, получили true, но забыли вызвать Commit – деньги как бы висят. В простом варианте A это сразу списано, нет проблемы. В варианте B важно, чтобы в коде, где резервируете, всегда был парный Commit (например, в BuildService).
* Data races нет (игра однопоточная), но логика может сталкиваться: например, одновременно ИИ и игрок пытаются резервировать гос-бюджет – этого у нас нет, т.к. бюджеты раздельные. Но вдруг решите сделать общий резерв – тогда нужны мьютексы или блокировки. В Unity MonoBehaviour контексте это редкость, но упомянуть полезно.

### Применение Codex GPT

* Реализация интерфейса: Спросите Codex: «Напиши интерфейс IEconomy с методами GetBalance, TryReserve, Commit, Tick». Он может сгенерировать код, останется скопировать.
* Класс Economy: Запрос «Реализуй класс Economy, который реализует IEconomy с базовым функционалом (баланс, проверка средств)». Codex, вероятно, предложит простой вариант с if(balance >= amount) { balance -= amount; return true; } для TryReserve. Проанализируйте, подходит ли вам эта модель (это вариант A). Вы можете уточнить: «А как реализовать отдельно резервирование и списание?», чтобы получить идею для варианта B.
* Генерация тестов: Очень полезно – попросите: «Придумай несколько unit-тестов для класса Economy, который умеет резервировать и списывать баланс». AI поможет сформулировать проверки инвариантов. Используйте это как шаблон, заменив на синтаксис NUnit (Assert.AreEqual и пр.).
* Логирование: Если не помните синтаксис Debug.LogWarning, можно спросить: «Как в Unity вывести предупреждение в лог». Он подскажет про Debug.LogWarning("text").
* Советы по дизайну: Задайте помощнику вопрос: «Лучше сразу списывать деньги при TryReserve или при Commit? Какие плюсы-минусы?». Возможно, он объяснит про транзакционность. Полезно для принятия решения.
* Расширенные механики: Если вы решите реализовать займы или контракты, можно проконсультироваться у Codex: «Как реализовать систему займов с процентами для игровой экономики?» – он может накидати идею формул. Или «Как спланировать бюджет с категориями (оборона, исследования)?» – тоже выдаст идеи, как хранить и обновлять.
* Отладка: Если тесты не сходятся, скормите Codex свой метод TryReserve/Commit и тест, который падает, с вопросом: «Почему этот тест не проходит?». Помощник проанализирует код и вероятно поймёт (например, вычитание не там стоит или сравнение float).
* Документация: Можно попросить сгенерировать XML-комментарии для IEconomy методов, чтобы сразу было описание, что они делают – мелочь, но приятная: «Добавь комментарии к методам IEconomy, объясни их назначение».

### Тестирование и проверка результата

* Unit-тесты: Убедитесь, что все разработанные тесты зелёные. Это формальный критерий, что базовая логика корректна. Попробуйте и негативные сценарии: искусственно вызвать Commit без TryReserve – ожидаете ли вы исключение или ничего? Если ничего – тоже хорошо, убедитесь, что так и есть.
* Интеграция с другими системами: Пока других систем мало, но хотя бы вручную попробуйте:
* Создайте временно кнопку UI, которая при клике пытается playerEconomy.TryReserve(50) и если true – playerEconomy.Commit(50). Запустите игру, нажмите. Посмотрите, баланс уменьшился ли на 50. Попробуйте ещё раз, пока не уйдёт в ноль и не начнёт отказывать. В логах должны быть сообщения об отказе после исчерпания средств.
* Если StateEconomy имеет пассивный доход, запустите игру без действий и посмотрите, растёт ли баланс гос-экономики с каждым тиком (при желании логируйте каждую секунду).
* Проверка в профайлере: Экономическая система сама по себе лёгкая, но гляньте профайлер – метод Economy.Tick должен занимать наносекунды. Важно, что он не генерирует сборок мусора каждый тик (не создаёт строки, списки и т.п. внутри цикла).
* Готовность к следующему модулю: К моменту завершения этого модуля, у вас должны быть:
* Два работающих объекта экономики (гос и игрок) с независимыми балансами.
* Возможность тратить из них деньги (методы TryReserve/Commit) и начислять доход (хотя бы у одного).
* Логи и тесты, подтверждающие, что деньги не пропадают бесследно и не появляются из воздуха.
* Эти объекты интегрированы в SimulationLoop.Tick (то есть каждый игровой тик экономика обновляется)[[31]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=%2F%2F%20%D0%B2%D1%8B%D0%B7%D1%8B%D0%B2%D0%B0%D0%B5%D0%BC%20%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%BD%D1%8B%D0%B5%20%D1%82%D0%B8%D0%BA%D0%B8%20%D0%BF%D0%BE,Flush%28%29%3B).
* Важно: Понимание, как дальше экономика будет использована – например, в следующем модуле (инфекция) напрямую нет, но в будущем: строительство объектов, реагирование ИИ на лимит бюджета. У вас должен быть готов интерфейс для этих взаимодействий. Если всё так – двигайтесь к следующему этапу.

## Модуль 4: Система инфекции (модель заражения на сетке)

### Краткое описание модуля

В данном модуле создадим систему распространения инфекции на игровой карте. Это ключевая механика симуляции – своего рода эпидемия, разливающаяся по клеточному полю. Мы реализуем упрощённую модель SIR (Susceptible-Infected-Recovered) или её вариацию, комбинируя её с клеточным автоматом на сетке. Каждая клетка может находиться в одном из состояний (восприимчивая, заражённая, иммунная) или иметь количественный уровень заражения[[34]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=5%29%20%D0%98%D0%BD%D1%84%D0%B5%D0%BA%D1%86%D0%B8%D1%8F%20%28%D0%BC%D0%BE%D0%B4%D0%B5%D0%BB%D1%8C%20%D0%B7%D0%B0%D1%80%D0%B0%D0%B6%D0%B5%D0%BD%D0%B8%D1%8F%29%20,%D0%B4%D0%BE%D0%BB%D1%8F%20%D0%BD%D0%B0%D1%81%D0%B5%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F%2F%D0%BC%D0%BE%D0%B1%D0%B8%D0%BB%D1%8C%D0%BD%D0%BE%D1%81%D1%82%D1%8C). На каждом шаге симуляции инфекция будет диффузно распространяться от заражённых клеток к здоровым соседям, с учётом некоторых параметров (например, базовая скорость заражения, плотность населения, возможно – наличие защитных сооружений).

Основной алгоритм: для каждой заражённой клетки повышаем уровень заражения соседних; если уровень превышает порог, соседние клетки переходят в заражённое состояние. Также учтём выздоровление/иммунитет: заражённые клетки со временем могут переходить в состояние R (Recovered) – что может означать иммунитет или вымирание инфекции там. Реализация будет двустадийной: сперва простая версия с полным перебором сетки, затем оптимизация – обновление только фронтира (границы заражения) и использование двойного буфера для корректности.

### Цели обучения

* Освоить концепцию клеточного автомата на примере эпидемической модели (SIR).
* Реализовать обновление сетки клеток по соседям (соседство Moore или Von Neumann) и понять разницу.
* Научиться работать с 2D массивами, эффективно обходить их и избегать ошибок индексов.
* Понять важность двойного буферизации при обновлении состояний, чтобы результаты текущего шага не влияли на вычисление соседей в том же шаге.
* Научиться визуализировать и логировать состояние поля для отладки (например, через Gizmos или цветовую карту).
* Подготовиться к оптимизации: понять, как можно обновлять только изменившиеся области (frontier) и вынести вычисления в многопоточные Jobs при необходимости.
* Закрепить использование Codex для генерации алгоритмического кода (например, перебор соседей) и отладки логики.

### Основные задачи (по возрастанию сложности)

1. Структура данных для поля. Решите, как будете хранить состояние инфекции для каждой клетки карты:
2. Если карта не слишком велика (например, 100x100), удобно завести двумерный массив или Grid<CellState>.
3. Создайте enum InfectionState { Susceptible, Infected, Recovered } для состояния S/I/R.
4. Создайте структуру CellInfection с полями: level (float от 0 до 1, степень заражённости) и state (тип InfectionState)[[35]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=,level%3B%20public%20InfectionState%20state%3B). В начале игры все клетки кроме, допустим, нескольких – Susceptible с level=0. Можно начальные очаги вручную задать или случайно.
5. Для доступа к соседям понадобится способ итерировать вокруг клетки: определите набор смещений для выбранного типа соседства:
   * Von Neumann (крест из 4 соседей): смещения (0,1), (0,-1), (1,0), (-1,0).
   * Moore (все 8 вокруг): дополнительно диагонали (1,1), (1,-1), (-1,1), (-1,-1).
6. Выберите один подход (например, Moore для более “круглого” распространения, или Von Neumann для более “крестообразного”). Зафиксируйте это и используйте везде одинаково[[36]](file://file_0000000069fc624685d94f067e42a757#:~:text=1,%D0%BA%D0%B0%D1%80%D1%82%D1%8B%3A%20%D0%BF%D0%BE%D0%BC%D0%BE%D0%B3%D0%B0%D0%B5%D1%82%20%C2%AB%D0%B3%D0%BB%D0%B0%D0%B7%D0%BE%D0%BC%C2%BB%20%D0%BB%D0%BE%D0%B2%D0%B8%D1%82%D1%8C%20%D1%81%D0%BA%D0%B0%D1%87%D0%BA%D0%B8%2F%D0%B7%D0%B0%D1%81%D1%82%D0%BE%D0%B8).
7. Создайте класс InfectionSystem с необходимыми полями: хранение сетки (например, CellInfection[,] grid), возможно ссылки на InfectionParams (порог, коэффициенты) из модуля 2.
8. Инициализация инфекции. Добавьте в начало игры (например, в GameDataLoader или отдельном InfectionSystem.Start) логику заполнения сетки:
9. Установите всем клеткам state = Susceptible, level = 0.
10. Определите 1-3 начальных заражённых клетки для старта сценария. Можно хардкодом: центр карты, или случайно выбрать несколько координат. У этих клеток state = Infected, level = 1 (максимум).
11. Возможно, при наличии PopulatonDensity или Mobility (из расширений) – пока опустим, сконцентрируемся на базовой модели.
12. Подключите InfectionSystem к SimulationLoop: в каждом тик вызов InfectionSystem.Tick(dt)[[7]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=while%20%28acc%20,Flush%28%29%3B).
13. Логика распространения (диффузия). Реализуйте метод Tick(float dt) в InfectionSystem:
14. Используйте двойной буфер: заведите второй массив или такой же размер CellInfection[,] newGrid, чтобы вычислять новые значения на основе старых, не переписывая старый на лету[[37]](file://file_0000000069fc624685d94f067e42a757#:~:text=%EE%88%80cite%EE%88%82turn4search0%EE%88%81%202,%D0%BA%D0%B0%D1%80%D1%82%D1%8B%3A%20%D0%BF%D0%BE%D0%BC%D0%BE%D0%B3%D0%B0%D0%B5%D1%82%20%C2%AB%D0%B3%D0%BB%D0%B0%D0%B7%D0%BE%D0%BC%C2%BB%20%D0%BB%D0%BE%D0%B2%D0%B8%D1%82%D1%8C%20%D1%81%D0%BA%D0%B0%D1%87%D0%BA%D0%B8%2F%D0%B7%D0%B0%D1%81%D1%82%D0%BE%D0%B8). После завершения расчёта, замените старую сетку новым (или скопируйте результаты).
15. Алгоритм одного тика:
    * Для каждой клетки (x,y) в старой сетке:
    * Если клетка Infected (заражённая):
      + Она сама может продолжать быть заражённой; можно реализовать выздоровление: например, уменьшить её уровень на небольшую величину (иммунизация, если хочется SEIR эффект). Либо оставить как есть, решив, что выздоровление дольше чем один тик – можно не делать тут, а делать по таймеру или отдельной фазой.
      + Рассмотрим всех соседей этой клетки (по выбранной схеме). Для каждого соседа, если он Susceptible, увеличиваем в newGrid его level на некоторую величину, зависящую от уровня текущей клетки и коэффициентов. Простой вариант: newLevel = oldNeighbor.level + infectRate \* dt. InfectRate возьмите из InfectionParams (например, 0.2).
      + Если после увеличения newLevel >= 1 (или какой-то порог), то в newGrid помечаем соседа как Infected (state) и можно задать newLevel = 1.
    * Если клетка Recovered: скорее всего, она не влияет на соседей (иммунная/пустая). Можно ничего не делать. Возможно, стоит слегка уменьшать уровень заражения (если осталось).
    * Если клетка Susceptible: по умолчанию ничего не делает сама, но её уровень мог быть повышен соседями (см. выше). Если после обработки всех заражённых соседей уровень остался <1, она остаётся Susceptible на этот тик.
    * Отдельно: можно вне цикла заразных соседей обработать выздоровление: для клеток Infected уменьшить уровень или, например, через какое-то время перевести в Recovered. Например, если клетка заражена X тиков, перевести её. Но для простоты: можете каждую заражённую понемногу снижать level, и если упадёт ниже 0.1 – перевести в Recovered.
16. После полного прохода замените grid = newGrid (предварительно можно старую сетку сохранить или копировать, если нужно сравнивать).
17. Логируйте суммарную статистику: например, посчитайте, сколько клеток стало Infected в этом тик, сколько выздоровело – выведите: "[Infection] Tick: newly infected N, recovered M, total infected K".
18. Проверьте, что не выходим за границы массива при обращении к соседям (не забудьте условия if(nx >=0 && nx < width && ny>=0 && ny<height)).
19. Визуализация и отладка поля. Чтобы понять, как распространяется инфекция, добавьте визуальные подсказки:
20. Gizmos: В скрипте InfectionSystem (или отдельном MonoBehaviour) реализуйте OnDrawGizmos() или OnDrawGizmosSelected(). В ней проходите по grid и рисуйте, например, квадраты или точки:
    * Gizmos.color = Color.Lerp(Color.green, Color.red, cell.level) для клетки, и Gizmos.DrawCube(position, new Vector3(1,0,1)) чтобы отобразить цветом. Однако Gizmos по умолчанию видны только в Scene View. Для отладки в редакторе это достаточно: вы будете видеть тепловую карту при запуске игры в окне Scene.
    * Отметьте несколько клеток: например, выбранную клетку и её соседей другим цветом, чтобы отладить логику соседства[[36]](file://file_0000000069fc624685d94f067e42a757#:~:text=1,%D0%BA%D0%B0%D1%80%D1%82%D1%8B%3A%20%D0%BF%D0%BE%D0%BC%D0%BE%D0%B3%D0%B0%D0%B5%D1%82%20%C2%AB%D0%B3%D0%BB%D0%B0%D0%B7%D0%BE%D0%BC%C2%BB%20%D0%BB%D0%BE%D0%B2%D0%B8%D1%82%D1%8C%20%D1%81%D0%BA%D0%B0%D1%87%D0%BA%D0%B8%2F%D0%B7%D0%B0%D1%81%D1%82%D0%BE%D0%B8). Например, если нажать на клетку (можно хранить какую-то debugSelectedCell), выделите её соседей: Gizmos.color = Color.cyan и нарисуйте WireCube вокруг соседних координат. Так вы убедитесь, что соседние индексы вычисляются правильно.
21. UI-оверлей: Более сложный вариант – отрисовать текстуру или Tilemap с цветовой заливкой поверх игрового поля. Если хотите, можете динамически обновлять цвет тайлов в зависимости от уровня заражения (Tilemap позволяет менять цвет отдельного тайла). Либо отрисовывать примитивы (например, Spawn sprite с прозрачным красным кружком в заражённых клетках). Это по желанию – можно ограничиться Gizmos, т.к. они уже дают картину в редакторе.
22. Логи деталей: Для глубокой отладки можно временно логировать изменения конкретной клетки: например, если уровень клетки (5,5) прыгнул с 0.2 до 0.5, вывести это. Но таких логов очень много на всю сетку, поэтому лучше делать их условно (например, if(x==5 && y==5) Debug.Log(...)).
23. Heatmap UI (опционально): если чувствуется уверенность, реализуйте в простейшем виде. Например, сделайте на Canvas картинку размером с карту, и манипулируйте её пикселями (Texture2D.SetPixel) по значениям level. Но это не обязательно.
24. Оптимизация обновления (frontier). По мере роста карты, обход всех клеток каждый тик может стать узким местом. Внедрите идею “frontier update” – обновлять только область вокруг текущего фронта заражения:
25. Поддерживайте список или набор координат, которые являются «активными» – где произошли изменения (например, недавно заражённые клетки, или клетки границы между заражёнными и здоровыми).
26. В каждом тик вместо полного двойного цикла по всей сетке, итерируйтесь только по списку активных клеток и их соседям[[38]](file://file_0000000069fc624685d94f067e42a757#:~:text=%E2%80%94%20%D1%82%D0%B5%D1%81%D1%82%20%C2%AB%D0%B7%D0%BD%D0%B0%D1%87%D0%B5%D0%BD%D0%B8%D0%B5%20%D0%BD%D0%B5%20%D0%B7%D0%B0%D1%82%D0%B8%D1%80%D0%B0%D0%B5%D1%82%D1%81%D1%8F,%D0%BA%D0%B0%D1%80%D1%82%D1%8B%3A%20%D0%BF%D0%BE%D0%BC%D0%BE%D0%B3%D0%B0%D0%B5%D1%82%20%C2%AB%D0%B3%D0%BB%D0%B0%D0%B7%D0%BE%D0%BC%C2%BB%20%D0%BB%D0%BE%D0%B2%D0%B8%D1%82%D1%8C%20%D1%81%D0%BA%D0%B0%D1%87%D0%BA%D0%B8%2F%D0%B7%D0%B0%D1%81%D1%82%D0%BE%D0%B8). То есть, если заражённая область компактна, вы обновите только её окрестности, а не весь мир.
27. Реализация: заведите List<(int x,int y)> frontierCells. Изначально – список начальных заражённых очагов. На каждом тик создайте новый список nextFrontier = []. Когда инфицируете соседей, добавляйте тех соседей, кто перешёл в Infected в этом тик, в nextFrontier (они станут фронтом на следующий шаг).
28. Также можно добавлять соседей, у которых уровень заметно изменился, даже если не заразились – но это тонкости. Минимум – новые заражённые и, возможно, клетки, которые стали здоровыми опять.
29. После обработки тик: frontierCells = nextFrontier.
30. Не забудьте, что если инфекция стихает где-то, фронт там пропадёт и процесс прекратится для той области (и это хорошо – не тратим время на пустые места).
31. Логируйте размер фронтира каждый тик и время обновления шага (можно замерять через Time.realtimeSinceStartup до/после Tick)[[38]](file://file_0000000069fc624685d94f067e42a757#:~:text=%E2%80%94%20%D1%82%D0%B5%D1%81%D1%82%20%C2%AB%D0%B7%D0%BD%D0%B0%D1%87%D0%B5%D0%BD%D0%B8%D0%B5%20%D0%BD%D0%B5%20%D0%B7%D0%B0%D1%82%D0%B8%D1%80%D0%B0%D0%B5%D1%82%D1%81%D1%8F,%D0%BA%D0%B0%D1%80%D1%82%D1%8B%3A%20%D0%BF%D0%BE%D0%BC%D0%BE%D0%B3%D0%B0%D0%B5%D1%82%20%C2%AB%D0%B3%D0%BB%D0%B0%D0%B7%D0%BE%D0%BC%C2%BB%20%D0%BB%D0%BE%D0%B2%D0%B8%D1%82%D1%8C%20%D1%81%D0%BA%D0%B0%D1%87%D0%BA%D0%B8%2F%D0%B7%D0%B0%D1%81%D1%82%D0%BE%D0%B8).
32. Проверьте, что если фронт пуст (никаких заражённых клеток) – то система не будет выполнять лишнюю работу (можно просто сразу выходить из Tick, инфекция закончилась).
33. Дополнительные механики (продвинутые, опционально):
34. Карантинные зоны: Позже, когда будет система строительства, можно предусмотреть, что определённые постройки (например, блокпосты) создают области, через которые заражение распространяется хуже или не распространяется вовсе. Реализовать это можно сейчас, если добавить в InfectionParams какой-нибудь коэффициент quarantineEffect, и в диффузии: если клетка сосед – карантинная (нужно знать, есть ли кордон на ней, а это значит, что система построек должна сообщать), то либо пропустить заражение, либо сильно ослабить (умножить на 0.1 например).
35. Очаги и фронт: Можно определить, что если уровень заражения клетки > 0.8, она считается “очагом”. Это чисто для статистики или для AI, но вы можете помечать такие клетки и, например, даже изменять стратегию распространения (например, очаги быстрее заражают вокруг).
36. Транспортные коридоры: если карта предусматривает дороги или порты – можно добавить, что некоторые клетки заражают дальние, не только соседей (например, вероятность переноса инфекции на большое расстояние). Но это уже усложнение, без которого модель тоже работает.
37. Случайность (стохастика): вместо детерминированного прироста можно вносить рандом. Например, шанс заражения соседа = 1 - (1 - p)^(количество заражённых соседей). Но для начального шага deterministic проще отлаживать. Случайность можно добавить позднее, убедившись в правильности базы.

### Пошаговые указания к реализации

* Создание InfectionSystem: В Scripts/Application создайте класс InfectionSystem. Он может не быть MonoBehaviour (можно делать обычный класс, который управляется SimulationLoop), либо MonoBehaviour без Update (раз вызывается вручную). В нём: поля ширины/высоты (можете взять из Tilemap size или зашить), двумерный массив CellInfection grid.
* Инициализация: В конструкторе или при первом вызове Tick проверьте, если grid не создан, то создайте массив нужного размера [width,height] и заполните Susceptible. Можно вынести в отдельный метод Initialize(width, height) и вызвать из GameManager при старте.
* Начальные очаги: Например, grid[50,50].state = Infected; grid[50,50].level = 1; frontierCells.Add((50,50)). Добавьте 2-3 таких для проверки, или сделайте функцию для заражения клетки (чтобы можно было и потом заражать искусственно).
* Tick реализация: Следуя описанному алгоритму, напишите двойной цикл. Можно оптимизировать, но сначала сделайте прямой полный обход, убедитесь, что работает.
* Создайте временный массив nextGrid того же размера. Лайфхак: можно копировать ссылки или MemberwiseClone, но проще пройтись по всем клеткам и скопировать. Либо изначально nextGrid = deep copy of grid, а потом изменять. В целях читаемости, сделайте двойной цикл: for x, for y – скопируйте значения. Затем ещё раз цикл – применяйте правила. (Или объедините, как удобнее, но следите, чтобы вы не перезаписывали, пока используете).
* Обход соседей: пропишите 4 или 8 проверок, или заведите массив neighbors = new int2[]{(1,0),(-1,0),(0,1),(0,-1), ...} и итерируйтесь по нему. Первый вариант явно длиннее код, второй – чище.
* Если используете frontierCells: тогда внешний цикл будет foreach(var (x,y) in frontierCells) и плюс возьмёте всех их соседей тоже. Но аккуратно: если у вас только заражённые клетки в фронтире, то соседи могут стать заражёнными – они войдут в nextFrontier. А что если заражённая клетка остаётся заражённой? Её тоже надо держать во фронте пока она имеет здоровых соседей. Так что, вероятно, и текущие заражённые тоже надо оставлять, пока вокруг них есть Susceptible.
* Простой подход: frontierCells хранит все заражённые клетки (Infected state) на границе с Susceptible. То есть, например, каждую симуляцию обновляйте frontier: собрать всех Infected, у которых есть хотя бы один сосед Susceptible. Это тоже допустимо, но требует сканировать соседей (но меньше, только вокруг заражённых).
* Начните без оптимизации (весь массив), убедитесь, что всё верно, потом добавьте frontier optimization: сравните результаты – они должны совпадать.
* Визуализация:
* Gizmos: override OnDrawGizmos in InfectionSystem (если не MonoBehaviour, можно сделать отдельный MonoBehaviour "InfectionVisualizer" и дать ему ссылку на InfectionSystem.grid). В Gizmos, делаете: for x,y => определяете цвет. Например,
* Color col;  
  switch(state):  
   case Susceptible: col = new Color(0,1,0, 0.1f + level\*0.5f); // зелёный, слегка краснеющий если уровень растёт  
   case Infected: col = Color.Lerp(Color.yellow, Color.red, level); // желто-красный  
   case Recovered: col = Color.gray;  
  Gizmos.color = col;  
  Gizmos.DrawCube(new Vector3(x, 0, y), new Vector3(1,0.1f,1));
* Это нарисует цветные кубики на сцене. Откройте сцену, включите Gizmos (в Game View Gizmos галочка, хотя лучше смотреть в Scene).
* Соседство отладка: если есть debugCellX, debugCellY (можно временно задать), выделите её Gizmos.color = Color.cyan, а всех соседей – Color.blue. Чтобы задать debugCell, можно, например, каждый тик брать первую заражённую клетку из frontier для анализа.
* UI overlay: если хотите, можете на Canvas вывести текст: количество заражённых, процент карты заражён. Либо, если реализовали debug menu, добавить туда переключатель отображения инфекции.
* Recovery: Добавьте простое правило: если клетка Infected, уменьшаем её level на, скажем, recoveryRate \* dt (например, 0.1 \* dt). Если level упало <= 0, или вы можете считать, что после 10 секунд заражения она переходит в Recovered. То есть, у вас может быть счётчик времени заражения. Но можно и проще: фиксированный шанс выздоровления. Чтобы сильно не усложнять – можете вообще не делать recovery, или сделать по таймеру: например, в CellInfection добавить поле infectionTime, в каждую Tick для Infected увеличивать его, и если > threshold, state = Recovered.
* Решите: Recovered клетки будут иметь level = 0? Или оставлять уровень как был? Наверное, лучше обнулить, или хранить, что они переболели (может влиять на AI).
* Межсистемная интеграция: Пока у нас нет эффекта экономики или построек на инфекцию, но имейте в виду: построенные игроком укрепления должны снижать распространение, требуя взаимодействия. Когда будет EventBus/BuildService, сделаем: при постройке, InfectionSystem узнает, что на клетке появился объект (например, фортификация), и меняет локально параметры (может, state = Recovered? или особый статус, или cell.protectionFactor).
* Подготовиться можно так: в CellInfection добавить поле hasFortification bool, и в распространении: если сосед.hasFortification, то либо пропустить заражение, либо умножить на фактор <1. Пока нигде не ставим true, но механика заложена.

### Логирование и отладка

* Проверка модели SIR: Убедитесь, что базовая «физика» SIR соблюдается[[39]](file://file_0000000069fc624685d94f067e42a757#:~:text=,%D1%80%D0%B0%D0%B7%D0%BC%D0%B5%D1%80%20%D1%84%D1%80%D0%BE%D0%BD%D1%82%D0%B0%20%D0%B8%20%D0%B2%D1%80%D0%B5%D0%BC%D1%8F%20%D1%88%D0%B0%D0%B3%D0%B0). То есть если вы настроили параметры (например, infectionRate, recoveryRate), попробуйте в небольшом примере на бумаге или в таблице прикинуть, как S->I->R должно происходить. Запустите симуляцию на маленькой сетке 3x3 и проследите, как одна заражённая клетка заражает соседей. Если уровень накапливается плавно, все хорошо. Если сразу все стали заражены – значит infectionRate слишком высок или threshold низкий.
* Соседство: Очень важно, чтобы вы реализовали соседей точно и не упустили никого. Проведите тест: выберите одну заражённую клетку, логируйте список её соседей координат. Сравните с тем, что вы ожидаете (например, у угловой клетки на краю должно быть 3 соседа в Moore, или 2 в Von Neumann). Вы можете для уверенности ручной тест сделать: например, функция, которая возвращает список соседей, и unit-тест, что для центра возвращает 8 (в Moore) уникальных, а для угла – 3.
* Двойной буфер: Проведите мысленный эксперимент или даже тест: если нет двойного буфера, то если вы обновляете сетку на лету, одна заражённая клетка может заразить соседа, а потом этот сосед в этом же тик заразит других – эффект «волны за один тик», чего быть не должно. С double buffer этого не случается – инфекция распространяется максимум на 1 клетку за тик, как задумано. Убедитесь, что вы правильно меняете newGrid, а старый читаете. Чтобы проверить:
* Можно создать ситуацию: центр заражён, вокруг 8 клеток незаражены. За один тик с double buffer заразятся только соседние 8 (если достаточно уровня), но не дальше. За второй тик – их соседи и т.д. Если сделать ошибку и обновлять старый сразу, может за один тик пойти дальше.
* Напишите Debug.Log при заражении клетки: "Cell (x,y) infected this tick". Запустите, посмотрите, не бывает ли случаев, что она стала зараженной, и тут же в том же тик вы логируете заражение кого-то, кто не сосед исходной, а через клетку (если infectionRate не огромный). Если так – ошибка в буфере.
* Frontier: Если реализовали обновление по фронту, сравните с наивным полным обновлением на каком-то шаге. Например, отключите frontier и запустите 50 тиков, сохраните состояние. Затем откатитесь, включите frontier-логику, запустите те же 50 тиков – состояния должны совпасть (проверить на паре клеток, или глобально подсчитать, например, общую зараженность). Если равны – оптимизация корректна.
* Отладка времени шага: Вывели время выполнения InfectionSystem.Tick (через Time или Stopwatch)? На маленьких размерах будет миллисекунды или меньше, но на больших (>100k клеток) может быть уже ощутимо. Не стремитесь к нулю пока, но держите в уме.
* Heatmap визуально: Откройте Scene View во время игры. Должны видеть картинку заражения. Убедитесь, что:
* В самом начале только несколько клеток красные (очаги).
* Спустя какое-то время образуется «пятно» заражения, распространяющееся плавно. Если видите скачки (например, внезапно дальняя клетка стала зараженной без цепочки) – вероятно, баг с индексами или не учли dt.
* Посмотрите, как infection level меняется: может понадобиться правильно настроить скорость. Слишком быстро – всё краснеет мгновенно; слишком медленно – распространение незаметно.
* Логирование прогресса: Выводите периодически (например, раз в секунду или раз в 10 тиков) процент заражённых клеток или количество текущих Infected. Это поможет убедиться, что модель не взорвалась или не заглохла. В SIR-модели обычно график: сначала экспоненциальный рост, потом насыщение и спад. Ваша игра – не обязательно реальная эпидемия, но похожие тренды могут быть.
* Частые ошибки:
* Неправильное вычисление соседних индексов (например, перепутали X и Y, или сделали neighbors массив неправильно). Это сразу видно по рисунку распространения – он будет искажён.
* Условие заражения: если вы сделали порог 1.0, но infectionRate\*dt никогда не достигает 1, может получиться, что level копится очень медленно и может не перейти в Infected статус (например, level=0.9 так и останется Susceptible). Чтобы этого избежать, можно порог сделать меньше, либо так: если level > 0, пусть это значит latent infection (еще S), пока <1. Но лучше: если достиг, сразу переводить state = Infected. Если не достиг, он сохранится и может накапливаться – это нормально.
* Не забыли ли обрабатывать Recovered? Иначе у вас никогда не убывает инфекция. Если recoveryRate=0, то Infected будут инфицировать вечно. Это ок, если игра про постоянную борьбу. Но если хотите реализовать спад, то нужно.
* Забытая проверка границ: если получаете IndexOutOfRange, добавьте проверки на краях.
* GC: Если каждый тик вы создаёте новый массив 100x100, это 10k objects GC может мусорить (на 60 FPS \* 10000 = 600k alloc/sec – не супер страшно, но лучше избегать). Можно сделать массив полем и переиспользовать, или, лучше, перейти на NativeArray (будет в оптимизации). На данном этапе, можно смириться, а оптимизировать в модуле 8.

### Применение Codex GPT

* Генерация кода распространения: Попробуйте запрос: «Алгоритм распространения инфекции на сетке 2D (Moore соседи) на C#». Codex может выдать псевдокод или даже близкий вариант. Используйте его идеи, но убедитесь, что код не содержит типичных ошибок (например, не обновляет сетку сразу).
* Двойной буфер: Спросите: «Почему нужен double-buffer при обновлении клеточного автомата?». Помощник объяснит и вы сможете убедиться, что делаете правильно.
* Работа с массивами: Если вам сложно с синтаксисом двумерных массивов или списков, Codex подскажет. «Как объявить двумерный массив 100x100 в C# и обойти его в цикле?» – это базово, но AI может быстро сгенерировать шаблон фор-циклов.
* Gizmos пример: Можно спросить: «Как через OnDrawGizmos нарисовать цветные кубики на каждой ячейке tilemap?». Codex, вероятно, выдаст пример использования Gizmos.DrawCube.
* Оптимизация frontier: Сформулируйте: «Как оптимизировать обновление клеточной автомата, обновляя только изменившиеся клетки?». Получите идеи насчёт frontiers. Вы можете даже попросить конкретно: «Напиши C# код, который хранит список изменённых клеток и обновляет только их соседей».
* Отладка: Если какая-то клетка ведёт себя странно (например, внезапно стала заражённой, хоть должна не сразу), можете описать ситуацию Codex: «У меня клеточный автомат, соседний алгоритм Moore. Заражённая клетка (5,5) за один шаг заразила (7,5), хотя между ними (6,5) не был заражён. Где искать ошибку?». Он может посоветовать проверить если вы не обновляете сетку по ходу (то есть укажет на double buffering).
* Визуализация: Если вы затрудняетесь с UI, можете спросить: «Как сделать тепловую карту заражения на Unity Tilemap?». AI подскажет про Tilemap.color или про Texture2D, это может дать идеи.
* Jobs/ECS на будущее: Codex можно спросить: «Как перенести обновление 2D grid в Unity Job System?». Он может дать пример использования IJobParallelFor по массиву – сохраните эту идею для модуля 8.

### Тестирование и проверка результата

* Малый сценарий: Создайте крайне маленькую сетку, 5x5 например, и один очаг. Пройдите тик за тиком вручную:
* Первый тик – очаг заражает своих (на 5x5 много крайних случаев – возьмите очаг не на краю для чистоты). Должны заразиться (или повыситься уровень) у соседей. Проверьте консоль/визуализацию.
* Второй тик – теперь бывшие соседи может заразят своих. Если double buffer работает, через одну клетку инфекция пока не должна проскакивать.
* Третий – должно дальше пойти. Если всё соответствует ожиданиям, хорошо.
* Большая карта: Запустите на, скажем, 50x50 с несколькими очагами. Проверьте:
* Инфекция распространяется примерно равномерно во все стороны от каждого очага (если параметры одинаковы).
* Два фронта встретятся, соединятся – в итоге получится одна область.
* Если сделали recovery, убедитесь, что через какое-то время рост замедляется и число заражённых стабилизируется или снижается.
* Наблюдение статистики: Если вывели логи, посмотрите: число новых заражений сначала растёт, потом падает – классическая кривая эпидемии. Если у вас recovery=0, то число заражённых будет только расти пока все не заразятся.
* Граничные эффекты: Проверьте, что на границах карты инфекция не вызывает ошибок и нормально останавливается (нет соседа вне массива).
* Проверка frontier оптимизации: Если включили, сравните скорость:
* Без frontier: можно измерить, сколько мс занимает Tick, когда, например, заражена половина карты.
* С frontier: когда заражена половина карты, фронтир – это в основном периметр области (примерно периметр ~ 2sqrt(N) vs NN если всё заражено, но если вся карта заражена, фронтир будет пуст).
* Убедитесь, что при полной заражённости вы не продолжаете обход (frontier пуст, Tick сразу выходит).
* Сравните лог "время шага" – с frontier оно должно быть меньше особенно на начальных стадиях, когда заражено мало клеток.
* Перед следующим модулем: Система инфекции должна работать автономно и правильно:
* Критерии готовности: Инфекция распространяется, видимо и предсказуемо; можно регулировать скорость и радиус через параметры (infectionRate, neighbors type); нет бесконечных багов (утечек, ошибок); debug-инструменты позволяют видеть состояние (вы можете глазами понять, что происходит, и AI в будущем тоже сможет).
* Теперь у нас есть динамическая часть симуляции (инфекция) и статическая (экономика). Далее займёмся связывающими элементами – события, постройки, команды, а также подготовим почву для ИИ.

## Модуль 5: Событийная шина и команды, система строительства и сохранение игры

### Краткое описание модуля

На этом этапе интегрируем механизм взаимодействия между различными частями игры с помощью EventBus (событийной шины) и системы команд. EventBus позволит рассылать сообщения о происходящих событиях (например, построен объект, начато исследование, произошёл всплеск инфекции) всем заинтересованным подсистемам без жёсткой связи между ними. Команды представляют собой действия, инициированные игроком или ИИ (например, команда построить башню обороны), которые можно логировать, откатывать или воспроизводить.

Мы реализуем простой шаблон EventBus с очередью и подписчиками[[40]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=%D0%9C%D0%B8%D0%BD%D0%B8,0%29). Затем создадим пример использования: система строительства (BuildService) как реализация команды. BuildService будет проверять возможность постройки (с учётом денег, занятости клетки), резервировать средства через Economy и публиковать событие BuildOrder в EventBus[[41]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=%D0%9A%D0%BE%D0%BC%D0%B0%D0%BD%D0%B4%D0%B0%20%D1%81%D1%82%D1%80%D0%BE%D0%B8%D1%82%D0%B5%D0%BB%D1%8C%D1%81%D1%82%D0%B2%D0%B0%3A%20,Publish%28order%29%3B%20return%20order). Подписчики (например, система карты/строений) получат это событие и непосредственно создадут объект на карте.

Дополнительно доработаем систему сохранения: убедимся, что она вызывается в правильный момент (после всех событий за тик, чтобы состояние консистентно)[[26]](file://file_0000000069fc624685d94f067e42a757#:~:text=%60,%EE%88%80cite%EE%88%82turn1search6%EE%88%81). Введём логирование команд для отладки (в будущем это поможет воспроизводить проблемы по логу).

### Цели обучения

* Освоить шаблон паблишер-сабскрайбер (Publisher/Subscriber) через глобальную шину событий для decoupling систем.
* Научиться определять классы событий и подписываться на них в разных системах.
* Реализовать безопасную рассылку событий после основного цикла симуляции (с помощью очереди событий).
* Понять роль команд: объект, инкапсулирующий действие. Научиться создавать и обрабатывать команды (например, команду на строительство).
* Интегрировать экономику с командами: проверка средств, списание при выполнении команды.
* Доделать систему сохранения: убедиться, что сохранение происходит вне тика, когда все события уже обработаны (тиковый барьер).
* Укрепить навыки логирования: лог каждого события/команды для отладки.
* Использовать Codex для генерации шаблонного кода EventBus, примеров команд и поиска ошибок подписки.

### Основные задачи (по возрастанию сложности)

1. Реализация EventBus. Создайте статический класс EventBus с методами:
2. Subscribe(object handler): регистрация подписчика. Подписчиком может быть любой объект, имеющий методы-обработчики событий. Проще всего – пусть подписчик реализует интерфейсы для событий или просто имеет публичные методы On(EventType e).
3. Publish<T>(T eventData): публикация события (объекта любого типа)[[40]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=%D0%9C%D0%B8%D0%BD%D0%B8,0%29). Реализуйте, чтобы событие складывалось в внутреннюю очередь q (Queue<object>).
4. Flush(): обработка всех накопленных событий. Пока очередь не пуста, доставайте событие и передавайте его всем подписчикам:
   * Самый простой вариант (из документа) – использовать dynamic: foreach(var h in subs) (h as dynamic).On((dynamic)e). Это позволит вызывать метод On у подписчика, соответствующий типу события. Например, если e типа BuildOrder, а подписчик имеет On(BuildOrder order), то dynamic-система найдёт его.
   * Если dynamic кажется сложным, альтернативно: можно завести интерфейс IEventHandler<T> для разных типов, но это добавит шаблонности. Для начала, dynamic – быстр и понятен, просто не забывайте, что нет проверки на этапе компиляции.
5. Храните списки подписчиков subs как List<object>[[42]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=public%20static%20class%20EventBus%20,Dequeue).
6. Добавьте защиту от исключений: если один подписчик кинет ошибку при обработке, лучше логировать, но продолжать остальные, чтобы EventBus не прервался.
7. EventBus должен вызываться в SimulationLoop.Tick последним шагом (вы уже сделали это в модуле 1). То есть все события, опубликованные во время тика, скопятся и только после обновления всех систем разошлются.
8. Подписка: вызовите EventBus.Subscribe(this) в Start/Awake тех систем, которые хотят слушать события. Например, BuildService может слушать команду строительства (хотя он сам её генерирует, здесь не нужно). А вот система, отвечающая за размещение зданий на карте (MapSystem) должна подписаться на BuildOrder события, чтобы создать объект.
9. Класс команды и событие строительства. Определите класс события/команды для строительства:
10. Например, public record BuildOrder(BuildingType Type, GridPos Position, Guid Who) – как в документе[[41]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=%D0%9A%D0%BE%D0%BC%D0%B0%D0%BD%D0%B4%D0%B0%20%D1%81%D1%82%D1%80%D0%BE%D0%B8%D1%82%D0%B5%D0%BB%D1%8C%D1%81%D1%82%D0%B2%D0%B0%3A%20,Publish%28order%29%3B%20return%20order), где Who или какой-то идентификатор, кто заказал (может быть игрок или ИИ).
11. Можно также добавить в BuildOrder поле результата (успешно/не успешно), но мы лучше будем публиковать только успешные заказы.
12. Определите enum BuildingType (например, Tower, Wall, Hospital… или просто для теста).
13. BuildService: создайте класс, реализующий интерфейс IBuildService[[43]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=public%20interface%20IBuildService%20,). В нём метод Place(BuildingType type, GridPos at, IEconomy whoPays):
    * Проверяет CanPlace(type, at, whoPays). Можно реализовать CanPlace просто: возвращает true, если на этой клетке ещё нет здания (нужно знать состояние карты) и хватает денег (whoPays.Balance >= cost).
    * Если нельзя, возвращает какой-то результат (можно сделать BuildOrder со статусом Failed или null).
    * Если можно: вызывает whoPays.TryReserve(cost). Если false – значит денег не хватило (возвращаем Fail).
    * Если резерв успешно: вызывает whoPays.Commit(cost) сразу (или можно публиковать событие и в подписчике списывать, но проще здесь).
    * Создаёт объект BuildOrder (с нужными данными, можно добавить например ссылку на Economy or just type/pos).
    * Публикует через EventBus: EventBus.Publish(order)[[44]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=public%20BuildOrder%20Place,Publish%28order%29%3B%20return%20order%3B).
    * Возвращает order (может не очень нужно, но на будущее).
14. Таким образом, BuildService отрабатывает синхронно при вызове Place, и реальный эффект (создание здания) произойдёт асинхронно, когда EventBus.Flush() доставит событие.
15. Подписчик события: решите, кто будет создавать сам объект игры (например, спрайт башни на карте). Логично, чтобы это сделал, например, класс BuildingManager или сам MapSystem. Сделайте простой MonoBehaviour BuildingPlacer который подпишется на BuildOrder. В нём метод On(BuildOrder order): там вы, например, можете нарисовать на Tilemap специальный тайл в этой клетке, или создать GameObject с соответствующим спрайтом. Пока можно просто логировать: Debug.Log($"Building placed: {order.Type} at {order.Position}") чтобы убедиться, что цепочка работает.
16. Хранение занятости клетки: Вам понадобится понимать, занята ли клетка уже зданием, чтобы CanPlace работал. Можно завести в InfectionSystem или отдельном GridManager флаг для клетки hasBuilding. Поскольку у нас Tilemap, можно использовать Tilemap to check (но лучше свой массив bool parallel to infectionGrid). На начальном этапе, можно хранить простейший Set<GridPos> occupiedBuildings.
17. Обновляйте этот Set, когда строение построено (в On(BuildOrder) добавьте позицию).
18. BuildService.CanPlace будет проверять, что !occupied.Contains(pos).
19. Рассылка других событий (опционально): Подумайте, какие события могут быть полезны:
20. Например, InfectionSpreadEvent: когда инфекция достигла определённого порога (скажем, новая область поражена). Вы можете публиковать событие, когда какая-то клетка заражается (но это много событий – лучше батчить, см. ниже).
21. ResearchStartedEvent: если бы была система технологий, при начале исследования.
22. BudgetChangedEvent: если экономика сильно изменилась (но обычно системы сами могут спросить).
23. Для практики, можно реализовать, например, событие: если за тик заражено > X клеток, публиковать OutbreakEvent(region). Это может слушать стратегический ИИ, чтобы реагировать.
24. Coalescing (склейка): Если много однотипных событий происходит одновременно, иногда их объединяют. Документ упоминает пример: "инфекция выросла в 50 клетках" – вместо 50 событий лучше одно с числом[[45]](file://file_00000000e2c461f4bb2cbf5057f4f2ac#:~:text=,50%20%D0%BA%D0%BB%D0%B5%D1%82%D0%BA%D0%B0%D1%85%C2%BB%20%E2%86%92%20%D0%BE%D0%B4%D0%B8%D0%BD%20%D0%B1%D0%B0%D1%82%D1%87). Если решите, реализуйте: собирать внутри InfectionSystem счетчик новых заражений, а после тик, если >0, публиковать один InfectionGrowthEvent(count) вместо множества.
25. Но эти доп. события не обязательны сейчас – главное отладить на примере BuildOrder.
26. Сохранение с учётом событий. Обновите логику сохранения (из модуля 2) так, чтобы:
27. Сохранение происходило после завершения тика, когда все события обработаны. В SimulationLoop у нас вызов EventBus.Flush() последним[[7]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=while%20%28acc%20,Flush%28%29%3B). Самый безопасный момент для сохранения – либо сразу после Flush (т.е. в конце Update()), либо вообще на паузе.
28. Если вы делаете сохранение по кнопке, возможно, она ставит паузу и затем вызывает Save – это отлично.
29. Главное: убедитесь, что вы не сохраняете в середине обработки событий или середине тика, иначе состояние может быть неполным (например, деньги списаны, а здание ещё не создано, или infectionGrid обновлён, а события о вспышках не разосланы).
30. Добавьте проверку: если Save вызывается по событию (например, UI), то либо ставьте паузу перед вызовом, либо внутри SaveLoadService, если SimulationLoop.IsPaused == false, сначала сделайте SetPaused(true), потом Save, потом можно вернуть обратно (или требовать, чтобы игрок нажимал паузу).
31. Тест: Постройте здание, в том же кадре сохраните, потом загрузите – здание должно присутствовать после загрузки. Если вы сохраняете до EventBus.Flush, то оно не будет записано (потому что ещё не создано). Значит, сохраняйте после flush.
32. Можно реализовать, что SaveLoadService подписывается на какое-нибудь специальное событие RequestSaveEvent и обрабатывает его после flush, делая Save. Тогда UI может просто Publish RequestSave, а SaveLoadService сделает на своём On.
33. Логирование и откладка команд. Для возможности анализа хода игры:
34. Сделайте список или лог буфер для команд: например, List<object> commandLog в EventBus или GameManager, куда добавляете все опубликованные события-команды. Можно ограничивать длину, чтобы не рос бесконечно.
35. Это поможет потом, если найдётся баг: посмотреть последовательность команд (например, игрок строил, строил, и что-то пошло не так).
36. Откат команд (undo): В нашем RTS в реальном времени, откат не обязателен, но упоминался как удобный механизм[[46]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=EventBus%20,%D1%81%D1%82%D1%80%D0%BE%D0%B8%D1%82%D0%B5%D0%BB%D1%8C%D1%81%D1%82%D0%B2%D0%BE%2C%20%D0%BD%D0%B0%D0%B7%D0%BD%D0%B0%D1%87%D0%B5%D0%BD%D0%B8%D0%B5%20%D0%B1%D1%8E%D0%B4%D0%B6%D0%B5%D1%82%D0%B0). Подумайте: вы можете реализовать, что каждая команда имеет метод Undo. Например, PlaceBuildingCmd.Undo мог бы снять здание и вернуть деньги. Пока это избыточно, но понимайте, что наличие команд логированых открывает эту возможность.
37. Playback: Сохраняя командный лог, можно воспроизводить игру с начала для реплея или отладки. Это за рамками текущей задачи, просто знайте, что командный буфер такой опцией может стать.
38. Логируйте EventBus.Flush: вы можете добавить в Flush вывод: [EventBus] Flushed (count=X)[[47]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=public%20static%20void%20Subscribe,), где X – сколько событий было обработано. Это поможет видеть, что каждый тик flush работал и сколько событий в нём было.
39. Логируйте каждое событие: возможно, не все, но важно, например, "[EventBus] BuildOrder processed for {Type} at {pos}" – либо в подписчике (BuildingPlacer) лог, либо в EventBus, когда доставляет (но лучше в подписчике, так виднее, кто обработал).
40. Дополнительное – ECS event buffer: В документе упоминается, что в ECS вместо EventBus используются EntityCommandBuffer для структурных изменений[[48]](file://file_00000000e2c461f4bb2cbf5057f4f2ac#:~:text=%2A%2A%D0%9A%D0%B0%D0%BA%20%D1%83%D1%81%D0%BA%D0%BE%D1%80%D0%B8%D1%82%D1%8C%3A%2A%2A%20,%D0%BF%D0%B0%D0%BA%D0%B5%D1%82%D0%BD%D0%B0%D1%8F%20%D0%BE%D0%B1%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%BA%D0%B0%2F%D1%84%D0%BB%D0%B0%D1%88%20%D0%BF%D0%BE%20%D0%BA%D0%BE%D0%BD%D1%86%D1%83%20%D1%82%D0%B8%D0%BA%D0%B0). Если решите потом частично перейти на ECS, вы будете подобный принцип использовать. Но сейчас не отвлекаемся, просто отметим: EventBus – аналог глобального посредника, в ECS – каждый систем имеет свой playback buffer.

### Пошаговые указания к реализации

* EventBus: Код можно писать почти по документу[[40]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=%D0%9C%D0%B8%D0%BD%D0%B8,0%29).
* Поля: static Queue<object> q = new Queue<object>(); static List<object> subs = new();.
* Subscribe(object handler) { subs.Add(handler); } – желательно проверять на дубли или отписку, но пока можно без.
* Publish<T>(T e) { q.Enqueue(e); }.
* Flush() { while(q.Count > 0) { var e = q.Dequeue(); foreach(var h in subs) { try { (h as dynamic).On((dynamic)e); } catch(Exception ex) { Debug.LogError($"EventBus error: {ex}"); } } } }
* Вызывать EventBus.Flush() в конце каждого SimulationLoop.Update (после цикла while(acc>=tick) ) – мы уже вставили вызов в модуле 1, убедитесь, что так и осталось[[7]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=while%20%28acc%20,Flush%28%29%3B).
* BuildService и команда:
* Создайте класс (можно статический) BuildService с методом, или инстанс (если нужно хранить что-то, но вроде нет).
* Сделайте enum BuildingType { Tower, ... } и struct/record GridPos(x,y) или просто используйте Vector2Int.
* Стоимость строительства: определите, откуда брать. Можно хранить в ScriptableObject (например, BuildingDef) – но вы еще не сделали BuildingDef. Для теста можно сделать в коде словарь cost: Tower -> 50, Wall -> 20. Или если в Data/Buildings/BuildingDef (SO) у вас есть, можно использовать: тогда BuildService надо знать BuildingDef по типу. Если вы сделали ScriptableObject BuildingDef with Id = "Tower" cost=50, то можно сделать Dictionary<string,BuildingDef> from module 2 load. Допустим, у BuildService есть доступ к GameDataLoader.buildingDefs or something. Но чтоб не сложно: можно захардкодить пару значений.
* Implement IBuildService (from arch doc) so that interface has methods CanPlace and Place[[43]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=public%20interface%20IBuildService%20,). Use them as described.
* Where to call BuildService.Place? – Это будет вызвано, например, из UI (когда игрок нажимает кнопку постройки и указывает куда) или из AI (когда AI решит строить). Пока у нас ни UI строительства, ни IИ команды. Для теста можем вызывать BuildService.Place прямо в коде (например, после 10 тиков автоматически, или по клавише). Можно, например, сделать: на клавишу "B" строить Tower в случайной клетке (or fixed cell).
* So for demonstration: in Update (or better in a DeveloperConsole or InputSystem) – if(Input.GetKeyDown(KeyCode.B)) { buildService.Place(Tower, somePosition, playerEconomy); } – This simulate player command.
* Подписчик BuildingPlacer:
* Create BuildingPlacer MonoBehaviour, put on some GameObject (e.g., on the Tilemap or GameManager). In Awake: EventBus.Subscribe(this).
* Implement public void On(BuildOrder order) – in it, implement the actual placement:
  + Mark the internal grid as occupied: occupiedBuildings.Add(order.Position).
  + If you have a Tilemap and tile for building, you can do: tilemap.SetTile(order.Position, buildingTile). But that requires setting up tile in editor.
  + For simplicity, instantiate a cube or sprite at that position: e.g., GameObject newTower = GameObject.CreatePrimitive(PrimitiveType.Cube); newTower.transform.position = new Vector3(x, 0, y); – That will show a cube where building is.
  + Or if 2D, create a small sprite of a house and place it.
  + If not comfortable with visuals, just log the placement.
* Also might want to inform other systems: for example, if building type influences infection (like fortification reduces infection spread in that cell), you could call infectionSystem.SetFortificationAt(order.Position, true) if you had such method. Could integrate: e.g., if order.Type is "Quarantine" or "Wall", you might reduce infection on neighbors. But that is advanced. For now, just note where building is so we don't allow another building same place.
* Saving buildings in GameState: Now that we introduced buildings, update GameState (from module 2) to include building data:
* E.g., have a list of built structures: List<BuildingData> where BuildingData has type and position.
* When saving, fill this list from your occupiedBuildings set.
* When loading, after restoring, you should recreate those buildings in the scene: e.g., call On(BuildOrder) for each or directly place them. Perhaps easier: after loading, loop through saved BuildingData and instantiate building objects (like how On would do).
* Alternatively, publish BuildOrder events during loading, but flush might need simulation loop – probably easier to directly place.
* Hook up Save/Load: If you have a UI, test saving after building something. Ensure load brings it back. If not, fix GameState to include necessary info (like building list).
* Testing event sequence:
* For thoroughness, simulate scenario: on same frame, player tries to build with insufficient funds:
  + BuildService will fail (money not reserved) and return maybe BuildOrder.Failed or null. We didn't define BuildOrder for fail. We can handle: if TryReserve fails, we do not Publish anything and return null.
  + So event not published, nothing happens – good.
* On success:
  + BuildService publishes event, flush at end of tick calls On in BuildingPlacer, building created.
  + Check money: playerEconomy should have been reduced by cost in BuildService at commit.
  + So after building, money down by cost, building appears.
  + If saving now, building is saved.

### Логирование и отладка

* Проверка EventBus последовательности: Добавьте временно лог внутри Flush: "Flushing event: " + e.GetType().Name. Так вы увидите в консоли, когда и какие события обрабатываются.
* Однократная доставка: Убедитесь, что каждый опубликованный event доставляется ровно один раз всем подписчикам:
* Например, вызвали две постройки в одном кадре: два события в очереди. Flush должен обработать оба.
* Проверьте, что после Flush очередь пустая и события не остаются на следующий кадр (по коду видно, что нет).
* Подписка/Отписка: Если у вас есть перезагрузка сцены, EventBus.subs остается статическим. Тогда старые подписчики могут накопиться. Простое решение: при стартовой инициализации (например, в GameManager.Awake) вызовите EventBus.subs.Clear(). Или, лучше, не делать EventBus static global across play sessions – but likely it is static for whole runtime. Just be cautious if writing Editor tests etc., clear between runs.
* BuildService:
* Проверьте CanPlace: test scenario where you attempt to place on the same cell twice – second time should fail due to occupied. Try it, it should not allow and likely no event published second time.
* Money check: if cost > balance, TryReserve returns false, Place returns fail (or null) – no event, building not placed. In console ideally see ReserveFail warning from Economy. That's expected.
* Event order: If you publish events from within events (not our case yet, but imagine On(BuildOrder) publishes another event), note that flush processes queue fully. So if On(BuildOrder) publishes, that new event will be queued and processed in the same flush after current loop or next loop? Actually, our implementation: while(q.Count>0) so if On publishes, that event goes to queue, and current while will catch it and process it in the same tick flush. That's fine usually. But if you want to delay to next tick, you'd have to design differently.
* For now, fine. But be aware: events published while flush is running will be processed immediately after the current event finishes, before returning to SimulationLoop (i.e., still within this tick).
* Сохранение проверка:
* Save the game after some events. Then simulate load. If something was mid-process, might not appear after load.
* For example, if you saved right after calling BuildService.Place but before flush (if that happened, maybe because Save was triggered in same tick), then your saved game wouldn't have the building. But we ensure Save after flush on UI anyway. So do that: test saving via a UI button that calls Save. If you click it immediately after building, since flush happens end of frame, maybe the button event came next frame. Eh, likely flush already done by the time UI events processed. If not sure, implement a slight delay or requirement to pause then save.
* Alternatively, call Save in On(BuildOrder) after placing building (not recommended, but could).
* Coalescing events: If you did implement something like grouping infection events, test it: are multiple similar events combined as intended? If yes, ensure flush still picks them up properly.
* Командный лог и отладка:
* If you made a list of commands, after some gameplay, print it out or inspect via debugger to ensure it recorded all events. Possibly skip this if not needed.
* If something goes wrong (e.g., building not appearing but money gone), check logs: Did BuildOrder event publish? Did flush run? Did On(BuildOrder) get called? Perhaps a typo in method signature (On(BuildOrder order) must match exactly the event type).
* Dynamic dispatch issues: If On signature mismatches (like parameter type not exactly the class, maybe you used struct vs class?), dynamic might not find method. E.g., if BuildOrder is record (class by default) and you pass record instance, it should match On(BuildOrder).
* If doesn't call, consider using where T: class in Publish and (h as IEventHandler<T>) check. But dynamic is easier albeit less safe.
* Частые ошибки:
* Забыли подписать на EventBus: ensure BuildingPlacer does Subscribe(this). If not, no placement happens.
* EventBus not flushed: ensure SimulationLoop calls Flush at end even if game paused? Actually, if paused, flush won't be called (because in Update we return early). If the game is paused, might want flush events from before pausing. But likely you'd pause after flush anyway. It's a corner case: if an event triggered exactly at pause, might remain queued. Could handle in SetPaused(true): flush events first.
* Race conditions of save: as above, but if careful, fine.
* Not thread-safe: EventBus is not thread safe, but our game is single-thread (except jobs, but they won't publish events directly hopefully).
* Overloading On: If multiple On methods (overloads) in one class, dynamic chooses best match or might be ambiguous. Avoid having e.g., On(BuildOrder) and On(object) in same class, dynamic might call both or the more generic one. Keep it distinct.

### Применение Codex GPT

* Генерация EventBus: Вы можете прямо запросить: «Реализуй класс EventBus на C# с методом Publish и Subscribe, использующий очередь для событий». AI наверняка выдаст нечто похожее на нужное.
* Использование dynamic: Если не уверены, спросите: «Как с помощью dynamic вызвать метод подписчика с подходящей сигнатурой?». Codex может объяснить или показать пример.
* Команда строительства: Попросите: «Напиши пример паттерна Command для строительства здания, с проверкой ресурсов». Возможно, получите код, аналогичный BuildService + команда.
* Взаимодействие с экономикой: Если сомневаетесь, правильно ли списываете деньги, спросите: «Как связать систему экономики и команду строительства в Unity?». Может подсказать, что вы уже сделали.
* Debugging: При сложных связях, можно описать Codex проблему: «Опубликовал событие, но подписчик не получает. Что проверить?». Он может подсказать проверить подписку, flush, method signature.
* Сохранение: «Как убедиться, что сохранение игры происходит после всех действий кадра?». Может предложить некоторые Unity patterns (like LateUpdate, etc., хотя EventBus flush already a kind of late update).
* Refactor: После всё работает, вы можете попросить AI улучшить код. «Предложи улучшения для EventBus реализации» – возможно, посоветует Generic approach. Но для учебных целей текущей хватит.

### Тестирование и проверка результата

* Функциональный тест строительства: В игре (Play Mode), попытайтесь построить:
* С достаточным бюджетом: объект появляется, баланс уменьшается ровно на стоимость.
* С недостаточным: ничего не происходит, в консоли предупреждение об отказе, баланс неизменён.
* На занятом месте: второй раз на ту же клетку – должен отказать (можно лог "CanPlace returned false").
* Сценарий с последовательностью: Постройте 2-3 разных объекта (если реализовали разные types):
* Убедитесь, что каждый раз корректно: места заняты, баланс суммарно уменьшен, объекты присутствуют (например, 3 кубика на сцене).
* Попробуйте сохранить игру и перезапустить (если загрузка реализована): все постройки должны восстановиться.
* Если есть InfectionSystem effect (вы не делали, но если хотели) – например, если башня должна снизить инфекцию вокруг, проверьте, что InfectionSystem получил информацию (setFortification flag).
* Проверка событий:
* В логах должен быть виден порядок: например:
* TickStart  
  ... systems update ...  
  TickEnd  
  [EventBus] Flushed (count=1)  
  [EventBus] BuildOrder processed for Tower at (10,5)
* Такого плана. Если flush count != events processed count, может значит, coalescing? But ideally one line per event from flush or subscriber.
* Нагрузите немного: publish 5 events in one tick (you can do that by calling BuildService.Place 5 times in one frame manually in code). Then flush should show count=5, and all 5 processed.
* Unit test approach: If possible, simulate events outside Unity:
  + Write simple test where you create dummy subscriber class with On(SomeEvent) that sets a flag. Publish event, call Flush, see flag true. If works, EventBus is correct.
* Multi subscriber: Add two subscribers for BuildOrder (e.g., building placer and maybe another logger object with On(BuildOrder) that just logs). Ensure both get called (like see two logs).
* Profiling: EventBus is negligible overhead unless thousands events. But check GC: dynamic invocation might allocate some, but usually it's minor. Not a big worry in this context.
* Перед следующим модулем: Итак, на данный момент у нас:
* Стабильная симуляция (время),
* данные/сохранения,
* экономика,
* инфекция,
* базовый механизм взаимодействия (события, команды) и пример строительства,
* UI/Dev controls (пусть и примитивные) для управления паузой, скоростью, постройкой, сохранением.

Критерии готовности: Игра работает как маленький, но цельный цикл: можно поставить на паузу, изменить скорость, вспышки инфекции происходят, можно тратить бюджет на строительство объектов, которые отображаются. В логах видно, что всё происходит в правильном порядке и данные можно сохранить/загрузить.

Теперь мы переходим к мозгу игры – модуль искусственного интеллекта, который будет принимать решения на основе всего этого.

## Модуль 6: Система ИИ (стратегический и тактический интеллект)

### Краткое описание модуля

Этот модуль посвящён разработке гибридной AI-системы для RTS, сочетающей несколько подходов: Utility AI для оценки приоритетов, GOAP (Goal-Oriented Action Planning) для планирования последовательностей действий и Behavior Trees для тактического поведения юнитов[[49]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=%D0%A0%D0%B5%D0%BA%D0%BE%D0%BC%D0%B5%D0%BD%D0%B4%D0%BE%D0%B2%D0%B0%D0%BD%20%D0%B3%D0%B8%D0%B1%D1%80%D0%B8%D0%B4%20%D1%82%D1%80%D1%91%D1%85%20%D0%BF%D0%BE%D0%B4%D1%85%D0%BE%D0%B4%D0%BE%D0%B2%3A). Такой гибрид позволит ИИ государства принимать осмысленные решения на стратегическом уровне (что важнее сейчас – сдерживать инфекцию, укреплять границы или инвестировать в исследования), выстраивать план для выбранной цели (например, для цели "сдержать очаг" – план из шагов: выделить бюджет, построить башни на периметре, отправить отряды) и управлять конкретными юнитами/отрядами на тактическом уровне (патрулирование, атака, ремонт).

Мы реализуем упрощённые версии этих систем: - Utility AI: набор критериев (considerations) с расчётом значимости разных возможных целей в виде нормализованных оценок 0..1[[50]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=1,%D0%94%D0%B5%D1%80%D0%B5%D0%B2%D1%8C%D1%8F%20%D1%85%D0%BE%D1%80%D0%BE%D1%88%D0%BE%20%D0%BE%D1%82%D0%BB%D0%B0%D0%B6%D0%B8%D0%B2%D0%B0%D1%8E%D1%82%D1%81%D1%8F%20%D0%B8%20%D1%87%D0%B8%D1%82%D0%B0%D0%B1%D0%B5%D0%BB%D1%8C%D0%BD%D1%8B). Например, оценка "сдерживать очаг инфекции" зависит от размера текущих вспышек; "строить укрепления" – от наличия уязвимых точек; "исследовать технологию" – от дефицита средств против заражения и т.д. На каждом стратегическом интервале ИИ будет выбирать цель с наивысшим скором. - GOAP: определим несколько простых действий (Action) с предусловиями и эффектами[[51]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=GOAP%20%28%D1%8D%D1%81%D0%BA%D0%B8%D0%B7%29%3A%20,Goal%20g%2C%20IWorldState%20w%29%3B), и одну-две цели (Goal). Построим план (можно вручную для простоты) или алгоритмически (поиск по состояниям) для выбранной цели. - Behavior Tree: смоделируем поведение отдельного юнита или группы. Используем простую структуру дерева решений: например, Selector с проверками угрозы, иначе патруль[[52]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=BT%20%28%D0%BF%D1%81%D0%B5%D0%B2%D0%B4%D0%BE%D0%BA%D0%BE%D0%B4%20%D1%83%D0%B7%D0%BB%D0%BE%D0%B2%29%3A%20,Patrol). Реализуем узлы BT как классы или просто как методы с состояниями (Running/Success/Failure).

Важно: сделаем слой восприятия (Perception), откуда AI получает агрегированные данные о мире[[53]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=%D0%A1%D0%BB%D0%BE%D0%B9%20%D0%B2%D0%BE%D1%81%D0%BF%D1%80%D0%B8%D1%8F%D1%82%D0%B8%D1%8F%3A%20,). Создадим интерфейс IWorldState или набор функций, чтобы AI мог узнать уровень инфекции в регионе, силу обороны на клетке, наличие chokepoint и т.п. Для начала, реализуем только пару методов (например, InfectionAt(position) и HasFortification(position)).

### Цели обучения

* Понять, как Utility AI помогает в выборе приоритета: научиться формулировать функции, оценивающие ситуацию и нормировать их на [0..1].
* Освоить принцип GOAP: задание целей, действий с предусловиями/эффектами, поиск плана. Даже если не полностью автоматизируем поиск, понять, как разбивать цель на действия.
* Разобраться в построении Behavior Tree: узлы-selector, последовательности, условия, действия. Научиться кодировать выполнение BT с сохранением состояния (какие ветки Running).
* Научиться соединять эти три подхода: Utility выбирает цель -> GOAP выдает план -> BT и/или скрипты выполняют конкретные шаги, плюс BT работает постоянно для реакции на окружение.
* Освоить техники логирования AI: печатать таблицу скорингов Utility[[54]](file://file_0000000069fc624685d94f067e42a757#:~:text=,%D1%80%D0%B5%D0%B6%D0%B8%D0%BC%D0%B5), логировать сгенерированный план GOAP[[55]](file://file_0000000069fc624685d94f067e42a757#:~:text=,%EE%88%80cite%EE%88%82turn2search2%EE%88%81), выводить активные узлы Behavior Tree[[56]](file://file_0000000069fc624685d94f067e42a757#:~:text=,%EE%88%80cite%EE%88%82turn2search5%EE%88%81), и визуализировать influence maps (если используем).
* Использовать Codex для помощи в алгоритмах (например, поиск плана) и генерации шаблонных классов (для BT узлов).
* Уделить внимание производительности AI: тиканье не каждый кадр, разделение вычислений по времени (budgeted AI)[[57]](file://file_00000000e2c461f4bb2cbf5057f4f2ac#:~:text=%2A%2A%D0%9A%D0%B0%D0%BA%20%D1%83%D1%81%D0%BA%D0%BE%D1%80%D0%B8%D1%82%D1%8C%3A%2A%2A%20,%D0%BE%D0%B1%D0%BD%D0%BE%D0%B2%D0%BB%D1%8F%D1%82%D1%8C%20%D1%82%D0%BE%D0%BB%D1%8C%D0%BA%D0%BE%20%D0%BF%D1%80%D0%B8%20%D0%BF%D0%B5%D1%80%D0%B5%D1%81%D1%82%D1%80%D0%BE%D0%B9%D0%BA%D0%B5%20%D0%BA%D0%B0%D1%80%D1%82%D1%8B).

### Основные задачи (по возрастанию сложности)

1. Utility AI – приоритеты стратегических задач. Определите 3–5 потенциальных целей для гос-ИИ:
2. Примеры: ContainOutbreak (сдержать заражение в очаге), FortifyChokepoint (укрепить узкий проход/границу), ResearchVaccine (исследовать технологию лечения), SupportPlayer (помочь игроку ресурсами или юнитами).
3. Для каждой цели разработайте функцию-оценку (Consideration), возвращающую значение 0..1 на основе текущего состояния мира:
   * ContainOutbreak: например, берём максимальный уровень заражения среди всех регионов или количество заражённых клеток. Нормируем: 0 – нет инфекции, 1 – инфекция вышла из-под контроля. Можно: score = clamp((infectedCellsCount - threshold) / (maxCells - threshold), 0,1). Или проще: score = infectedCellsCount / totalCells (доля карты заражена).
   * FortifyChokepoint: надо определить, есть ли "горлышки" (chokepoints). Можно грубо: если заражение близко к базе или к важной точке (например, столице) и там мало обороны. Определите условно, что если какой-то регион со стратегическим значением и он под угрозой, score высокий. Или просто наличие незакрытых проходов на границе безопасной зоны.
   * ResearchVaccine: если инфекция растёт, а у вас нет технологий – повышается. Может зависеть от кол-ва доступных ресурсов (если ресурсов много, можно инвестировать).
   * SupportPlayer: если у игрока мало денег или ситуация у игрока плохая (например, потерял много строений), гос-ИИ может помочь. Пока сложно симулировать, но можно: если playerEconomy balance << stateEconomy balance, то score для поддержки.
   * Influence Maps: Создайте пару полей влияния для помощи в расчетах[[58]](file://file_00000000e2c461f4bb2cbf5057f4f2ac#:~:text=,%D0%B8%20%D0%B1%D0%BE%D0%B3%D0%B0%D1%82%D0%B0%20%D0%BD%D0%B0%20%D1%8D%D0%B2%D1%80%D0%B8%D1%81%D1%82%D0%B8%D0%BA%D0%B8%20%EE%88%80cite%EE%88%82turn1search1%EE%88%81):
   * enemyPressure[x,y] – например, уровень инфекции или врагов в данной точке (можно взять infectionGrid normalized).
   * friendlyStrength[x,y] – число оборонительных сооружений или юнитов (пока, скажем, 1 для каждой башни).
   * Можно вычислить агрегаты: например, функция FriendlyStrengthIn(region) – считает количество башен в этом регионе. InfectionAt(pos) – уже есть в InfectionSystem.
   * Реализуйте каждую Consideration как класс, реализующий интерфейс IConsideration с методом Score(IWorldState)[[59]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=Utility,5f%3B). Эти классы могут хранить параметры (например, порог threshold). Например, OutbreakPressureConsideration – см. документ пример.
4. Создайте класс StrategicAI (можно MonoBehaviour, можно чистый), который хранит список целей и соответствующих Consideration. Например, можно просто хранить Dictionary<Goal, IConsideration>.
5. Каждые N тиков (не каждый, чтобы не перегружать) – например, раз в 10 тиков – StrategicAI обновляет оценки:
   * Для каждой цели вычисляет score = consideration.Score(worldState). Логирует эти значения (табличкой или JSON)[[54]](file://file_0000000069fc624685d94f067e42a757#:~:text=,%D1%80%D0%B5%D0%B6%D0%B8%D0%BC%D0%B5).
   * Выбирает цель с максимальным скором. Если она уже текущая цель и план по ней выполняется – можно оставаться. Если другая стала значительно выше – можно переключить цель (завершив предыдущую или прервав, в реальности).
   * На данном этапе, для простоты, можно сразу переключать на новую топ-цель каждый интервал, чтобы посмотреть разные реакции.
6. GOAP – планирование действий для выбранной цели. Определите набор Action (действий), которые ИИ может предпринять:
7. Действия можно привязать к целям. Например, для цели "ContainOutbreak":
   * Action1: AllocateBudget – увеличить резерв на оборону/карантин на X (предусловие: бюджет > X, эффект: оборонный ресурс увеличен).
   * Action2: BuildTowers – построить Y башен на рубеже (предусловие: есть свободные позиции на фронте, есть бюджет, эффект: укрепления на фронте).
   * Action3: DeployUnits – отправить отряд (если бы у нас были юниты).
   * Для "ResearchVaccine":
   * Action: StartResearch – потратить деньги и время на технологию (предусловие: есть нерешённая технология, эффект: технология начата/завершена).
   * Для "SupportPlayer":
   * Action: SendFunds – перевести часть бюджета игроку (предусловие: баланс гос > X, эффект: баланс игрока +X, баланс гос -X).
8. Реализуйте интерфейс IAction с методами: Preconditions(IWorldState), Execute() (или Apply) и Cost()[[51]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=GOAP%20%28%D1%8D%D1%81%D0%BA%D0%B8%D0%B7%29%3A%20,Goal%20g%2C%20IWorldState%20w%29%3B).
   * Preconditions: проверяет, можно ли выполнить (например, для BuildTowers: есть ли незакрытые chokepoints).
   * Execute: собственно, делает действие (в нашем случае, может быть: вызывает BuildService.PlaceTower на конкретной клетке, или просто publishes Event "DeployUnits").
   * Cost: для планирования, можно возвращать ресурсную стоимость, или время. GOAP планер обычно ищет минимальный cost план. У нас небольшие деревья, можно не заморачиваться, просто возвращать, например, 1 для кажд. действия, или actual money cost.
9. Planner: Для демонстрации, не обязательно писать универсальный A\* planner. Можно захардкодить мини-план: например, если цель = ContainOutbreak, план = [AllocateBudget, BuildTowers]. Если цель = SupportPlayer, план = [SendFunds]. И т.п. Но лучше попробовать автоматический:
   * Можно реализовать очень упрощённый поиск:
   * Начальное состояние: текущий мир (например, availableBudget = X, towersBuilt = false, etc., abstract).
   * Цель: условие, которое нужно достичь (например, infectionContained = true).
   * У действий укажите эффекты (можно просто булевые, e.g., BuildTowers -> towersBuilt=true).
   * Тогда планер (BFS/DFS) будет пытаться применить действия, удовлетворяющие предусловия, и смотреть, достигается ли цель.
   * Для учебных целей, можно не погружаться глубоко: покажите концепцию – у вас есть цепочка действий, которые логируются как план.
10. В StrategicAI, после выбора цели, вызывайте Planner.Plan(goal). Полученный список действий сохраните как текущий план.
11. Логируйте план: например, [AI] Plan for ContainOutbreak: AllocateBudget -> BuildTowers -> ...[[55]](file://file_0000000069fc624685d94f067e42a757#:~:text=,%EE%88%80cite%EE%88%82turn2search2%EE%88%81).
12. Выполнение плана: Можно выполнять по шагам:
    * Завести, например, переменную currentActionIndex. Каждые N тиков (или каждый тик) проверять: если текущий план не пуст:
    * Если предусловия текущего действия выполнены – выполняем (например, вызываем Execute()) и переходим к следующему.
    * Если не выполнены – может, нужно сначала что-то (в реальном GOAP, порядок гарантирует, но если что-то поменялось, можно перепланировать).
    * После выполнения последнего действия, цель считается достигнутой или цикл заново оценит Utility и выберет новую.
    * Можно упростить: сразу выполнить все действия подряд (но если они должны распределяться по времени, лучше растянуть). Для стратегических – можно несколько тиков, но не критично.
13. Привяжите реальные системы:
    * Например, AllocateBudget – может просто log, т.к. у нас нет детализации бюджета по статьям (или можно уменьшить playerEconomy, но зачем). Можно skip actual effect, или simulate by storing a variable "defenseBudget += X".
    * BuildTowers – здесь можно интегрироваться: найти позиции на фронте инфекции (можно взять пару клеток, где infection level just at border), и вызвать BuildService.Place(Tower, pos, stateEconomy). Это круто, если сработает: ИИ сам строит башни. Если денег хватит – Economy уже в системе.
    * SendFunds – implement as: stateEconomy.TryReserve(X) and if true, playerEconomy.Commit(X) (basically transfer). Or simpler, directly do stateEconomy.Balance-=X; playerEconomy.Balance+=X. Also publish event for UI if any.
    * StartResearch – if had techs, could toggle a tech as researched. For now, maybe skip actual effect.
14. Behavior Trees – тактическая логика. Если у нас будут юниты (в RTS обычно есть), Behavior Tree управляет их действиями. Пока у нас нет реализованных юнитов, но можем смоделировать на примере ремонта башен или патруля:
15. Допустим, у игрока/ИИ есть рабочий или солдат (представим один объект). Он может: патрулировать, чинить постройки, реагировать на угрозу.
16. Построим BT:

* Selector  
   ├─ If(ThreatNear) -> Sequence( MoveToThreat, Attack )  
   ├─ If(DamagedBuildingExists) -> Sequence( MoveToBuilding, Repair )  
   └─ Patrol
  + В псевдокоде выше: проверяется, есть ли угроза рядом (например, инфекция прорвалась в безопасную зону). Если да – прервать рутину и "отправить отряд" (в игре: spawn unit or move existing), но у нас нет боёв, можно просто лог: "Unit would attack threat at X".
  + Иначе, если есть повреждённое здание (для примера, представим, что заражение может "повреждать" постройки, или просто случайно сделаем здание нуждающимся в ремонте): тогда юнит идёт туда и "чинит".
  + Иначе – патрулирует (можно лог или перемещать по маршруту).

1. Как реализовать:
   * Сделайте класс BehaviorTree с узлами. Узлы могут быть разными типами: Selector, Sequence, Condition, Action, etc.
   * Для простоты, можно не делать целую иерархию, а в коде BehaviorTree.Tick() прописать логика:
   * if(threatNear) {  
      // perform attack sequence  
      ...  
      return;  
     }  
     if(damagedBuilding) {  
      // perform move & repair  
      ...  
      return;  
     }  
     // else  
     patrol();
   * Это не чистый BT с state though.
   * Лучше: Implement Node base class, and composite classes for Selector, Sequence, etc.
   * Each node has a method Execute() returning enum Status { Running, Success, Failure }.
   * Selector: iterate children until one returns Success/Running.
   * Sequence: iterate children until one fails (like typical BT).
   * Condition node: returns Success if condition true else Failure (and often immediate).
   * Action node: executes some action (like move or attack) and returns Success if done, Running if ongoing (like moving takes time).
   * Memory for Running status: If an action is Running (e.g., unit moving, not finished in one tick), we need to preserve that state and call its Execute again next tick.
   * Real BT frameworks have node state or the tree is ticked each frame and nodes maintain state in fields.
   * We can simulate Running by e.g., splitting an action into multiple ticks or using a coroutine concept. But to keep it simple:
   * For demonstration, we might not simulate actual long actions, just logs. Or we can simulate "movement" by a counter that counts ticks to reach target.
   * If doing the structure is too heavy, we can fallback to simpler logic or pseudocode demonstration in logs.
2. Attach BehaviorTree to some "unit" (maybe we consider the player has a mobile unit).
3. Integrate with event system if possible: e.g., if threat appears, maybe publish an event that unit picks up. But let's keep it internal to BehaviorTree check for now.
4. Logging:
   * Each tick, log which branch executed (e.g., "Threat detected -> Attack sequence running" or "No threats -> patrolling")[[56]](file://file_0000000069fc624685d94f067e42a757#:~:text=,%EE%88%80cite%EE%88%82turn2search5%EE%88%81).
   * If we had node IDs or names, log active path (some dev tool print).
   * Keep logs minimal to not spam but enough to debug logic.
5. For demonstration: you can simulate a scenario:
   * Infect some area near base (set threatNear=true by checking if any infected cell within some radius of base coordinate).
   * Then BehaviorTree should choose Attack branch (log it).
   * If no infection, but we mark a building as damaged (simulate a building needing repair by a variable), then it should go repair branch.
   * If neither, patrol (log "patrolling...").
6. Though we do not have actual game objects moving, you can incorporate the idea. Maybe if time, instantiate a "unit" GameObject that moves (like moves one step per tick to target pos) to illustrate Running state.
7. BehaviorTree tick frequency: likely every tick for unit AI, as tactical decisions can be quick. If worried performance, can do every couple ticks for heavy tasks, but a single unit is trivial.
8. Перцепция и Influence maps. Implement part of IWorldState (or directly integrate in Utility) to provide the AI the info it needs:
9. Provide methods in StrategicAI or a WorldState class:
   * InfectionAt(GridPos p): returns infection level at that cell (this can just read from InfectionSystem grid)[[53]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=%D0%A1%D0%BB%D0%BE%D0%B9%20%D0%B2%D0%BE%D1%81%D0%BF%D1%80%D0%B8%D1%8F%D1%82%D0%B8%D1%8F%3A%20,).
   * HasFortification(GridPos p): check if there's a building/tower at p (we track occupiedBuildings)[[53]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=%D0%A1%D0%BB%D0%BE%D0%B9%20%D0%B2%D0%BE%D1%81%D0%BF%D1%80%D0%B8%D1%8F%D1%82%D0%B8%D1%8F%3A%20,).
   * FriendlyStrengthIn(Region r): If you define "region" (maybe chunk of map or around base), compute sum of friendly units/buildings. For now, maybe define one region "base" and count all towers or a constant if any tower present.
   * IsChokePoint(GridPos p): we may not have calculation for that; but maybe we can set manually certain coordinates as chokepoints (like a narrow pass coordinate). Or simply, define a couple positions and pretend they are passes. Or don't use it to avoid complexity.
10. Influence maps:
    * If you want to be fancy: compute an array infectionInfluence[x,y] as some diffusion of infection presence (which might just be the infection levels themselves).
    * But doc2 suggests incremental update of influence and caching, which is advanced. We might skip heavy computing and say influence = infection level or a blurred version.
    * Friendly influence: maybe for towers, mark influence radius around them for defensive influence (like each tower influences nearby 5 cells).
    * These influences can be used by Utility scorers (e.g., if a region shows high infection influence and low friendly influence, the score for contain outbreak is high).
    * Implementation: you could do a pass on infection grid and create an influence map (like convolution).
    * E.g., for each infected cell add value to neighbors at distance (maybe like a Gaussian blur).
    * But if time short, you can skip actual calc and approximate with counting infected vs towers as we do in Utility.
11. Ensure that what Utility or BT asks is provided by these worldstate functions (e.g., BehaviorTree threat detection: "threat near base" might mean check infectionInfluence at base > some threshold).
12. Provide worldState to considerations (they implement Score(IWorldState)).
    * This means StrategicAI should have or be IWorldState or have reference to systems to implement IWorldState methods.
    * Could implement IWorldState in StrategicAI, where StrategicAI has references to InfectionSystem, building list, etc.
13. Check that these queries are efficient: our map is not huge so fine, but in general would consider caching heavy ones.
14. AI integration and execution. Tie everything together:
15. Create an object StrategicAI (the state brain) and maybe a TacticalAI if separate (or incorporate BehaviorTree within StrategicAI for now).
16. Tick frequency:
    * Strategic decisions (Utility evaluation and GOAP planning) we do less often, e.g., once per second or every 10 ticks[[57]](file://file_00000000e2c461f4bb2cbf5057f4f2ac#:~:text=%2A%2A%D0%9A%D0%B0%D0%BA%20%D1%83%D1%81%D0%BA%D0%BE%D1%80%D0%B8%D1%82%D1%8C%3A%2A%2A%20,%D0%BE%D0%B1%D0%BD%D0%BE%D0%B2%D0%BB%D1%8F%D1%82%D1%8C%20%D1%82%D0%BE%D0%BB%D1%8C%D0%BA%D0%BE%20%D0%BF%D1%80%D0%B8%20%D0%BF%D0%B5%D1%80%D0%B5%D1%81%D1%82%D1%80%D0%BE%D0%B9%D0%BA%D0%B5%20%D0%BA%D0%B0%D1%80%D1%82%D1%8B). We can achieve by counting ticks in SimulationLoop or StrategicAI itself (like only act when tickCount % 10 == 0).
    * Tactical Behavior we can run more frequently (maybe each tick or every 2 ticks).
    * Document suggests making strategic layer tick less frequently than tactical[[57]](file://file_00000000e2c461f4bb2cbf5057f4f2ac#:~:text=%2A%2A%D0%9A%D0%B0%D0%BA%20%D1%83%D1%81%D0%BA%D0%BE%D1%80%D0%B8%D1%82%D1%8C%3A%2A%2A%20,%D0%BE%D0%B1%D0%BD%D0%BE%D0%B2%D0%BB%D1%8F%D1%82%D1%8C%20%D1%82%D0%BE%D0%BB%D1%8C%D0%BA%D0%BE%20%D0%BF%D1%80%D0%B8%20%D0%BF%D0%B5%D1%80%D0%B5%D1%81%D1%82%D1%80%D0%BE%D0%B9%D0%BA%D0%B5%20%D0%BA%D0%B0%D1%80%D1%82%D1%8B).
17. In SimulationLoop, add calls:
    * e.g., every tick: TacticalAI.Tick(dt),
    * every 10th tick: StrategicAI.Tick(dt) (which internally does Utility and possibly triggers actions).
18. Or incorporate as separate MonoBehaviours scheduled differently. But easier to manage from SimulationLoop.
19. Confirm orders:
    * If StrategicAI triggers a BuildService.Place, that event flows through event bus and building appears (so AI physically builds stuff).
    * If it triggers sending funds, that directly alters balances.
    * If research or other, track logically.
20. After setting up, run the simulation:
    * Start with some infection scenario and some starting budgets.
    * Let AI run through, see which goal picks:
    * If infection present, likely ContainOutbreak highest, triggers plan to build towers (money permitting) etc.
    * If infection small and base safe, maybe Research or something chosen.
    * Without infection, perhaps picks SupportPlayer or idle if none triggered.
    * Check logs for Utility scores, plan chosen, plan executed, Behavior tree acting possibly.
    * See results: Are towers placed where needed? Are finances adjusting?
    * Possibly you'll see multiple towers built (draining economy) if not regulated by cost or by plan stops after one.
    * You could add that after building once, outbreak contained (score goes down next evaluation).
    * Behavior Tree:
    * Try simulate an infection approaching base to see if BT logs "attack threat".
    * For building repair, if none integrated, you might simulate a building "damaged" by just marking a variable. Then see if BT picks it up.
21. Debug overlays:
    * If possible, use Unity Gizmos or UI to show influence maps:
    * E.g., in OnDrawGizmos, draw heatmap of infectionInfluence in another color vs friendlyInfluence.
    * Or at least draw lines or icons: could draw a blue sphere at position of highest infection (like marking outbreak).
    * Or draw text above towers showing "HP" if any concept of damage.
    * For BT, you could show a line from unit to target if moving to repair or attack (if we had actual coordinates).
    * Possibly more advanced than needed, but helpful if you can illustrate e.g. which chokepoints considered or where towers built.
22. Performance:
    * Our AI logic is light due to small scale, but note:
    * Utility evals reading some arrays - fine.
    * GOAP planning trivial here.
    * Behavior tree one unit trivial.
    * But if many units, BT per unit could add up, one might budget them (like update different units in different ticks).
    * Also utility can be split (some scorers this tick, others next) to avoid spike (that's the "budgeted AI" concept).
    * We won't see performance issues at this scale, but log if you want how long Utility calc took. Possibly integrate with profiler for curiosity.
23. Advanced improvements (optional):
24. Incremental Utility: If some scores are heavy (like scanning whole map for infection count) every time, you could cache these results or update incrementally each tick (e.g., track infected count in InfectionSystem, then just use that number).
25. Learning weights: Usually Utility can have weights or curves. We can assume all normalized and equally important for now. Or manually adjust if one always overshadow others (like if infection present, likely that always top).
26. Communication: If we had multiple agents (like allied AI, or multiple factions), could have events or shared knowledge. Not needed here.
27. Adapting to player: The SupportPlayer goal is a bit stand-in for something interactive (like if player struggling, AI help). Hard to simulate beyond just transferring money. Could skip if not meaningful now.
28. Multiple units: If wanted to simulate a small squad, you could spawn few dummy units and each runs a copy of BehaviorTree with slightly different tasks (or one BehaviorTree controlling group).
29. UI for AI: Perhaps final overlay showing current AI goal and plan ("AI currently focusing on Outbreak containment") etc. This can help debug but not required.

### Пошаговые указания к реализации

* Utility Considerations:
* Create classes like ContainOutbreakConsideration, FortifyConsideration, etc., each deriving from IConsideration (with float Score(IWorldState)).
* Implement Score: you have worldState accessible. For ContainOutbreak: maybe float infectedRatio = worldState.TotalInfected() / worldState.TotalCells(); return Mathf.Clamp01((infectedRatio - 0.1f)/0.5f); as an example. Or simpler: return worldState.TotalInfected() > someThreshold ? 1 : worldState.TotalInfected()/threshold;.
* For Fortify: maybe check if any choke point (pretend we have one at some coordinates) is adjacent to infected area and no fortification there: if yes, return 1, else 0. This could be binary or gradually: if infection is within X distance of base and no towers in between, high score.
* For Research: maybe based on time or just default medium if others low.
* For SupportPlayer: if playerBalance < some threshold and stateBalance high, then moderate-high score.
* We'll likely tune by observation: the relative scales might cause one to always dominate. Possibly fine if infection always triggers that path, because that is logically pressing.
* After implement each, gather them in StrategicAI.
* StrategicAI:
* Fields: list of goals (could be just an enum or string names) and corresponding Consideration, and maybe an active plan.
* Possibly define a small class or struct for Goal: with name, maybe a target location if needed (like "fortify at (X)" if applicable).
  + Actually, some goals naturally have target (outbreak location, chokepoint coordinate). We can incorporate in planning stage: e.g., if ContainOutbreak chosen, find coordinates of worst outbreak region and pass to plan actions (like build towers around that region).
  + Could store that as part of goal selection: like maintain GridPos targetOutbreakCenter.
  + For simplicity, perhaps pick the first infected cell or the one with highest level as target for containment. Determine that in Score function or after choosing goal.
  + Implementation: maybe have worldState provide GetLargestOutbreakCenter() returning a pos. Then StrategicAI can pass that to BuildService or others.
* Method Tick:
  + if tickCount % 10 !=0, maybe skip (so only do every 10).
  + Compute all scores:
  + float bestScore = -1; Goal bestGoal = null;  
    foreach(var (goal, consideration) in goals) {  
     float s = consideration.Score(this /\*assuming this implements IWorldState\*/);  
     Log($"[Utility] {goal.Name} score = {s:F2}");  
     if(s > bestScore) { bestScore = s; bestGoal = goal; }  
    }
  + If bestGoal != currentGoal or if no currentPlan or plan done:
  + currentGoal = bestGoal.
  + Plan plan = Planner.Plan(bestGoal).
  + currentPlan = plan (list of actions).
  + currentActionIndex = 0.
  + Log ("Selected new goal: X, planned actions: ...").
  + Else if bestGoal same as currentGoal, perhaps continue existing plan, unless replan needed (if something changed drastically, you could replan, but skip complexity).
  + Then handle executing plan:
  + If currentPlan not null:
    - If currentActionIndex < plan.Count:
    - var action = plan[currentActionIndex];
    - If action.Preconditions(worldState) are met:
      * Do action (some actions might be instantaneous like sending funds, some might involve waiting like building might take time but we treat as immediate after calling BuildService).
      * Log action executed.
      * currentActionIndex++.
    - else:
      * If precond fails (maybe world changed so plan invalid?), could replan or skip. For simplicity, re-check preconditions rarely fails since we planned on current world state. We might ignore this scenario or just replan entire goal if fails.
    - If currentActionIndex >= plan.Count:
    - Plan done. Possibly log "Goal achieved or plan completed".
    - Could either set currentGoal = null to allow new selection, or keep it and rely on utility next tick to maybe switch (if still top, maybe have to come up with a new plan or maintain).
    - Possibly some goals are continuous (like containOutbreak might never "finish" until infection gone). You could then plan again next time with maybe same actions if needed.
    - We can set plan = null to force new utility evaluation next cycle (which happens anyway).
  + It's possible multiple actions in plan should not all be done instantly in one tick (like building multiple towers might better do one per tick or so).
  + We can choose to execute one action per strategic tick call. That might naturally spread by the tick frequency:
  + Because if StrategicAI.Tick is only called every 10 ticks, then each call executes one action of plan. That spreads them out.
  + But our plan likely short and one time actions, so either is fine.
  + Let's choose to do at most 1 action per call to StrategicAI.Tick for realism. Use a loop break after one.
  + That means a plan might take multiple cycles to complete, which is okay.
* Ensure to incorporate the effect:
  + If BuildService called, event bus will handle building (so possibly by next tick towers exist).
  + If multiple towers to build, plan might list action BuildTower three times. If we do one at a time per call, it will build one tower per 10 ticks for example.
  + If find that too slow in demonstration, you can allow more frequent or multiple actions per tick for brevity.
* Planner:
* Could implement a trivial one:
  + If goal == ContainOutbreak: plan = [ new BuildTowerAction(target= worldState.GetLargestOutbreakEdge()) ] (maybe allocate budget action if needed, but if economy handled cost anyway, not needed separate).
  + If goal == FortifyChokepoint: plan = [ new BuildTowerAction(target = chokepointPos) ] (like identify one known chokepoint coordinate).
  + If goal == SupportPlayer: plan = [ new SendFundsAction(amount = maybe some portion of stateEconomy.Balance ) ].
  + If goal == Research: plan = [ new ResearchAction(tech = some pre-defined tech) ].
  + If already built and research done, maybe do nothing next time.
  + This is more like a rule-based plan than algorithmic GOAP, but given limited scenario, it's fine.
* If implementing generic:
  + Make simple state representation:
  + A state could have booleans like hasFortified, hasResearched, and values like money.
  + Goals define desired booleans true.
  + Actions define which booleans they set and what they require true/false.
  + Then DFS until find all goal conditions satisfied (with some limit).
  + But might be overkill. We'll do the rule approach to ensure something plausible happens.
* Action classes:
* For each action:
  + e.g., BuildTowerAction : IAction:
  + Has maybe a target GridPos or Region it will build in.
  + Precondition: worldState.HasFortification(target) is false AND worldState.InfectionAt(target) > 0 maybe (meaning it's at frontier).
  + Execute: call BuildService.Place(Tower, target, stateEconomy). If returns successful (BuildOrder event), consider it done.
  + Could also directly place building if wanting immediate, but better to go through BuildService to use event flow and economy.
  + This action might fail if not enough money (which BuildService would handle internally). Possibly we should check economy before calling or catch if returns fail (lack money).
  + If fail, maybe mark precond false next tick or re-evaluate plan. But keep it simple: assume enough funds (we can ensure by not choosing to build if not money by Utility or by adding an AllocateBudget before).
  + SendFundsAction : IAction:
  + Precondition: stateEconomy.Balance >= X.
  + Execute: stateEconomy.TryReserve(X) & Commit, playerEconomy.Commit(X). Or we could create an event and do it via EventBus, but direct is fine.
  + It's effectively done immediately.
  + ResearchAction : IAction:
  + Precondition: some tech not researched.
  + Execute: perhaps reduce economy balance by cost (like spending money) and mark tech done (set a bool).
  + This action mainly to demonstrate could tie to an actual future effect like unlocking something, but here just conceptual.
  + AllocateBudgetAction:
  + Precondition: none (or maybe if we had internal budgets).
  + Execute: simply log or adjust an internal field "defenseBudget += amount".
  + This could be used if building towers needed to allocate money ahead, but since our economy is just one pool, not needed really. Possibly skip implementing this, or implement but it doesn't do anything real.
* Add these classes under StrategicAI or separate files accordingly.
* Behavior Tree Implementation:
* Suppose you have one unit to manage (maybe state or player unit).
* Create a class UnitAI with a BehaviorTree or equivalent logic in Update.
* If you want to simulate movement:
  + For Attack: we don't have actual enemies, could simulate by printing "Engaging threat at (x,y)" and maybe after 3 ticks say "Threat neutralized".
  + For Repair: if you want, you can mark a building "damaged" and then "repair it" by some action that could increase its HP. But since we don't simulate damage, perhaps have an event or trigger to simulate that.
  + Actually, we could simulate building damage due to infection:
  + If an infected cell is near a building, mark that building as damaged or destroyed after some time (just as scenario).
  + Then BehaviorTree sees damaged building, goes to repair (which could just log "Repairing building at X, done").
  + Patrol: maybe just log "Patrolling area" each time if nothing else to do.
* Implementation can be as simple as:
* if(threatNearBase) {  
   // Attack sequence (simulate time maybe)  
   if(not already moving/attacking) { Log "Unit heading to threat"; set state "attacking" and maybe store target; }  
   else { // in progress  
   // after a couple ticks, assume done:  
   if(counter++ > someValue) { Log "Threat eliminated"; state = Idle; counter=0; }  
   }  
  } else if(damagedBuildingExists) {  
   if(state != "repair") { Log "Heading to repair building X"; state="repair"; }  
   else {  
   if(counter++ > someValue) { Log "Building repaired"; state = Idle; counter=0; mark building repaired; }  
   }  
  } else {  
   if(state != "patrol") { Log "Patrolling..."; state="patrol"; /\*maybe set a random patrol point\*/ }  
   // we could toggle patrol state or just log repeatedly  
  }
  + This is more state machine-like, but can mimic a BT's behaviour.
  + Alternatively, code a simple BT structure:
  + Define objects for conditions and actions:
    - Condition threatNear: returns true if any infection within e.g., 10 tiles of base.
    - Action AttackThreat: if not at threat location, "move" (increment a step), else "attack" (simulate done).
    - With small environment, easier to stick to state machine approach as above.
  + For demonstration clarity, a state machine is fine, as long as we log decisions similarly to BT logic (because conceptually it's same branches).
* If more ambition: implement Node classes and a root node that you tick:
  + But due to time, the state machine approach is acceptable. If logs clearly show the fallback logic, it suffices.
* One more integration: Behavior tree could also be influenced by events:
  + E.g., if EventBus published an AttackCommand, the unit would go attack. But here threat detection is automatic from environment.
  + Or if building got an Event "BuildingDamaged", unit responds. That might be more realistic:
  + Could do: when infection infects a cell with building, publish BuildingDamaged event. UnitAI subscribes and on event sets state to go repair that building. That would be interesting demonstration of event interplay.
  + If time: possibly simulate that:
    - in InfectionSystem, if an infected cell neighbors a tower, publish BuildingDamagedEvent(pos).
    - UnitAI On(BuildingDamagedEvent e) => drop whatever and set target to e.position for repair.
    - That would emulate reactive behaviour triggered by events, which is often how it can be done (subscribe to events in BT).
  + But without actual infection damaging code, you could artificially fire such an event by pressing a key or at some time to test.
  + This might complicate but adds show of event usage beyond building.
  + Up to whether implement given time, but even just logging with check is fine.
* AI testing scenario:
* At start of game, maybe spawn an infection cluster not too far from a defined "base" area.
* Place maybe one initial tower (simulate base defense).
* Provide stateEconomy some funds (like 200).
* Provide playerEconomy maybe some too (like 50, though if support triggers they get more).
* Then run:
  + Infect spreads, Utility likely picks ContainOutbreak with high score. Plan: build tower(s). Should call BuildService => towers appear around infection hopefully (depending how we pick location).
  + If build location selection not implemented (we might default to just build at a preset position near outbreak).
  + Could simply do: target = one of infected cell's coordinates for building. But building on infected cell not realistic, maybe a neighbor. But either is demonstration.
  + Or do: find frontier cell (Susceptible neighbor to infected) and build there. But time may be short for that logic, can cheat by building at infected cell itself to show something built.
  + After building, infection might be "contained" in a sense (no, it will still spread unless building had effect which we did not implement).
  + We could incorporate an effect: if tower present at a cell, maybe infection doesn't spread beyond? If so, that's actual containment. But we didn't implement infection effect of fortifications. We could:
    - Quick fix: in InfectionSystem.Tick, if a cell has tower (fortification), reduce neighbor spread by factor for those directions.
    - For demonstration, maybe not needed, just building stands as if defense.
  + If stateEconomy becomes low after building some towers, maybe SupportPlayer might take priority (if we gave a condition like if no outbreak or if they think player needs help).
  + If infection ended or slowed, maybe support triggers. If not, support might always be lower because outbreak still pressing.
  + Behavior Tree unit:
  + if infection near base triggered, logs "attacking threat at (pos)" a few times then "threat eliminated".
  + If we simulate building damage, see logs "repairing building at X".
  + If nothing, "patrolling".
  + Check that these transitions happen logically.
  + Watch economy values after building: money should reduce accordingly.
  + Possibly the AI might overspend and not have enough for further actions (like if towers are expensive). That's fine, then maybe plan actions will fail if no money (we should log if BuildService returns fail).
  + Could add logic: if BuildService returned fail, log "Not enough funds, cannot execute BuildTower".
  + Maybe next utility evaluation might then decide something else if can't build.
  + Save/Load to ensure the new towers, any changes persist (though AI states may not fully persist in current design unless we saved internal states, but not needed if Save soon after building etc).
  + Also verify our logs coverage:
  + Utility scores printed occasionally,
  + Plan logged,
  + Each executed action logged,
  + Behavior tree actions logged.
  + That gives a nice trace of the AI's thinking and acting.

### Логирование и отладка

* Utility scores table: Check that all scores appear and make sense. For example:
* [Utility] ContainOutbreak = 0.75  
  [Utility] FortifyChokepoint = 0.20  
  [Utility] ResearchVaccine = 0.10  
  [Utility] SupportPlayer = 0.05  
  Chosen goal: ContainOutbreak
* If any score is 0 or 1 constantly, might need adjusting calculation (maybe saturating).
* If always one goal is maxed, others never chosen (which might be fine if outbreak persists). But test scenario with no infection: do we then see research or support become top? If no infection, containOutbreak score should be 0, others might have some baseline.
* Possibly define baseline or default so AI just doesn't idle if no pressing issues: e.g., research could have baseline 0.2 always if not done. So in times of peace, it picks research.
* Plan log: After goal selection, you should see something like:
* [AI] Plan for ContainOutbreak: BuildTower(x=...,y=...) -> (maybe multiple if you did)
* Or if plan more than one action:
* [AI] Plan for SupportPlayer: SendFunds(50)
* Action execution: Each executed should log:
* [AI] Executing BuildTower at (10,12)
* Then separate logs from event system:
* [Economy] ReserveFail... (if fail)  
  [EventBus] Flushed (count=1)  
  [EventBus] BuildOrder processed for Tower at (10,12)
* So you can correlate AI's intent with actual outcome.
* Behavior logs:
* [UnitAI] Threat detected near base -> engaging  
  [UnitAI] Threat eliminated.
* or
* [UnitAI] Repairing tower at (8,8)  
  [UnitAI] Tower repaired.
* or "Patrolling...".
* Ensure the conditions trigger properly:
  + Simulate threatNear by maybe setting one infection at base coordinates (like InfectSystem infect base, causing threshold cross).
  + After "eliminated", maybe you remove that infection cell to simulate cleared.
  + If building repair triggered by event or check, ensure you simulate a building's "health" status variable.
* If using event for building damage:
  + Infect a cell adjacent to a tower position -> in InfectSystem you can check each infected cell, if any building within 1 tile, publish BuildingDamaged.
  + That event triggers UnitAI On(BuildingDamagedEvent) which sets a target for repair.
  + Then logs "Heading to repair ...".
  + If multiple events, unit might get spam. Could handle only one at a time (e.g., ignore new if already busy).
  + It's fine just to show one scenario.
* Timing:
* Confirm StrategicAI tick frequency working: you can print tick count in logs to see that it only prints scores every 10 ticks, not every tick (ensures we aren't doing heavy calc every tick).
* Behavior every tick likely, fine.
* Utility fragmentation: If one score needed heavy calc, could do e.g., compute outbreak data on tick5 and fortify on tick6 etc. But probably unnecessary now.
* Correctness:
* See that towers built by AI indeed at least attempt to be placed where needed.
* If plan had multiple towers but economy only enough for one, after first, economy might not allow second:
  + If your plan tries second immediately, BuildService will fail (money).
  + If one action per tick, then on second tick it fails, maybe you catch that and might re-evaluate plan next cycle (since likely containOutbreak still high but now it might plan differently or realize budget low).
  + We didn't code adaptively for fail, but as long as logs note fail, it's okay. Could improve: if build fails due lack funds, maybe set some cooldown or reduce containOutbreak score (since can't do anyway without money).
  + Not needed to fix, just acknowledge if appears in log.
* Edge conditions:
* If infection goes to 0 (like after some time in simulation if you manually clear it or in test scenario you stop it):
  + Then containOutbreak score should drop to 0, AI picks next goal (maybe research).
  + This switch should show in logs at next utility evaluation.
* If infection goes extremely high:
  + Score saturates at 1, always picks that. That is fine, AI will keep trying same goal. Possibly building more and more towers until funds end.
  + If that happens, might observe multiple towers placed. Good to see event bus coping with multiple events. Should be fine.
* Performance:
* With our small numbers, likely negligible. Just ensure no endless loops or something if plan logic had mistakes.
* If we attempt pathfinding later (module 7), note that if many path queries from AI could be heavy, but we haven't integrated pathfinding yet here.

### Применение Codex GPT

* Utility & Scoring:
* Ask for advice: «Examples of Utility AI considerations for RTS?». Possibly get ideas like "if enemy units > threshold then defense score high", etc. Adapt to infection context.
* «How to normalize values for Utility AI scoring?» might yield formulas or best practices (like using logistic curves, etc.). For now, linear clamp is fine.
* GOAP:
* «Simple GOAP implementation in C#» may produce code. There are known examples. Ours is simplified but reading those could inspire how to structure.
* «GOAP FEAR Orkin summary» might give description which matches doc mention[[60]](file://file_00000000e2c461f4bb2cbf5057f4f2ac#:~:text=,%D0%B8%20%D0%B1%D0%BE%D0%B3%D0%B0%D1%82%D0%B0%20%D0%BD%D0%B0%20%D1%8D%D0%B2%D1%80%D0%B8%D1%81%D1%82%D0%B8%D0%BA%D0%B8%20%EE%88%80cite%EE%88%82turn1search1%EE%88%81). But we won't go that deep now.
* Behavior Tree:
* «C# Behavior Tree example» could produce an implementation outline. Useful if we wanted to implement classes Node, etc. Possibly too heavy to do fully correct now, but check:
* Actually, doc snippet suggests an example structure, but not an implementation. Codex might produce a quick BT code. Might be instructive but possibly too large to integrate completely now.
* We can still attempt for curiosity.
  + If it's short and understandable, incorporate piece or two.
  + E.g., «Implement a simple behavior tree with Selector and Sequence nodes in C#» - let's see what it suggests.

## Модуль 7: Поиск пути (Pathfinding) на карте

### Краткое описание модуля

В реальном времени юниты или агенты игры должны уметь находить путь по сетке (тайловой карте) – обходить препятствия (например, заражённые зоны, горы, стены) и выбирать кратчайший маршрут. В этом модуле мы реализуем алгоритм A (A-star) – стандарт для поиска кратчайшего пути в сетках – и рассмотрим расширения: ускорение поиска через Jump Point Search (JPS) на сетке и идею иерархического поиска (HPA)\*\* для больших карт[[61]](file://file_00000000e2c461f4bb2cbf5057f4f2ac#:~:text=,%D0%BF%D1%80%D0%BE%D1%85%D0%BE%D0%B4%20%D1%87%D0%B5%D1%80%D0%B5%D0%B7%20%D0%B7%D0%B0%D1%80%D0%B0%D0%B6%D1%91%D0%BD%D0%BD%D1%8B%D0%B5%20%C2%AB%D0%BA%D1%80%D0%B0%D1%81%D0%BD%D1%8B%D0%B5%C2%BB%20%D0%B7%D0%BE%D0%BD%D1%8B).

Базовый A обеспечит корректный поиск пути, а JPS оптимизирует его на больших открытых областях за счёт пропуска промежуточных узлов. HPA (Hierarchical Pathfinding) – это продвинутый подход, где карта разбивается на кластеры, и поиск ведётся сначала между кластерами, затем уточняется внутри – он значительно ускоряет навигацию на огромных картах[[61]](file://file_00000000e2c461f4bb2cbf5057f4f2ac#:~:text=,%D0%BF%D1%80%D0%BE%D1%85%D0%BE%D0%B4%20%D1%87%D0%B5%D1%80%D0%B5%D0%B7%20%D0%B7%D0%B0%D1%80%D0%B0%D0%B6%D1%91%D0%BD%D0%BD%D1%8B%D0%B5%20%C2%AB%D0%BA%D1%80%D0%B0%D1%81%D0%BD%D1%8B%D0%B5%C2%BB%20%D0%B7%D0%BE%D0%BD%D1%8B). Мы реализуем A и, при желании, JPS; HPA опишем концептуально для понимания.

### Цели обучения

* Научиться представлять игровое поле для алгоритмов pathfinding: задать проходимость клеток (walkable/blocked), веса (стоимость) и эвристику.
* Реализовать алгоритм A\* поиска пути: работа с множествами open/closed, расчёт стоимости пути, эвристической функции (например, манхэттеново расстояние) и восстановление найденного маршрута.
* Понять ограничения A и области его оптимизации. Освоить Jump Point Search – улучшение A для равномерных сеток, уменьшающее количество проверяемых узлов за счёт «прыжков» по прямым линиям.
* Понять концепцию иерархического pathfinding (HPA\*): научиться разбивать карту на области и прокладывать маршрут сначала на абстрактном уровне областей, а затем детализировать – хотя бы на уровне мысли, без полного кода.
* Научиться отлаживать алгоритмы поиска: визуализировать открытые/закрытые узлы, финальный путь, сравнивать длины путей от разных алгоритмов[[62]](file://file_0000000069fc624685d94f067e42a757#:~:text=,%EE%88%80cite%EE%88%82turn3search5%EE%88%82turn3search20%EE%88%81).
* Интегрировать поиск пути в игру: уметь вызывать функцию pathfinding для перемещения юнита или рассчитывать расстояния; если есть время – использовать pathfinding в поведении ИИ (например, чтобы тактический ИИ мог двигаться к цели).
* Освоить использование Codex для анализа сложного алгоритма: можно просить объяснить код A\*, помочь оптимизировать его или найти ошибку (например, если путь странно прокладывается).

### Основные задачи (по возрастанию сложности)

1. Представление сетки и препятствий. Подготовьте структуру данных для карты:
2. Если у вас уже есть Tilemap в Unity, можно использовать координаты тайлов. Заведите, например, двумерный массив bool walkable[width,height] или подобную структуру, где true означает, что по клетке можно пройти, а false – препятствие.
3. Заполните этот массив на старте: отметьте границы карты как непроходимые, клетки с зданиями (башнями, стенами) как непроходимые. Можно также решить, считать ли заражённые клетки препятствиями: реалистично юниты могут через них проходить, но с риском. Для начала пометьте инфекцию как проходимую, но можно задать ей повышенную стоимость (например, чтобы путь через заражение был менее желателен). В простейшем случае используем бинарную сетку (проходимо/нет); усложнить можно позже добавлением весов.
4. Определитесь с соседством для движения: обычно 8-смежное (включая диагонали) на сетке. A\* на 8-соседях должен учитывать, что движение по диагонали чуть дороже (длина √2 ≈ 1.414). Для упрощения можно разрешить только 4 направления (вверх/вниз/влево/вправо) – тогда движение по диагонали возможно лишь в два хода. Мы выберем 8-направлений, но с учётом стоимости: по горизонтали/вертикали – 1, по диагонали – √2 (можно хранить эти значения как 10 и 14, если работаем с целыми для эффективности).
5. Определите функцию эвристики для A\*: обычно берут манхэттеновское расстояние (|dx|+|dy|) для 4-напр. движения или диагональную эвристику (которая учитывает и диагонали) для 8-направлений. Манхэттенская часто подходит и для 8 направлений, но слегка недооценивает путь; можно использовать h = 10 \* max(|dx|, |dy|) если 10 — цена шага по горизонтали, тогда учитываем диагонали.
6. Реализация алгоритма A\*. Напишите функцию FindPath(start, goal) возвращающую список координат от старта до цели:
7. Структуры данных: создайте класс/структуру Node с полями: координаты (x,y), gCost (стоимость от старта), hCost (эвристика до цели), fCost = g+h, а также ссылку parent (для восстановления маршрута). Также понадобится коллекция open set (например, List<Node> или лучше приоритетная очередь) и closed set (список или bool-массив отмечающий закрытые клетки).
8. Инициализация: поместите стартовую клетку в open set (g=0, h рассчитан).
9. Цикл поиска: пока open set не пуст:
   1. Взять узел с наименьшим fCost (и hCost при равенстве) из open set (вручную найти или использовать приоритетную очередь).
   2. Если это целевая клетка, нашли путь – выход.
   3. Переместить этот узел в closed set.
   4. Рассмотреть всех его соседей (до 8). Для каждого соседа:
      * Если сосед вне границ или !walkable, пропустить.
      * Если сосед уже в closed set, пропустить (значит, мы его уже оптимально обработали).
      * Вычислить предполагаемую новую gCost = current.gCost + стоимость шага (10 или 14).
      * Если сосед не в open set, или новая gCost меньше старой (нашли более короткий путь к соседу):
      * Обновить параметры соседа: gCost = новая, hCost (по эвристике), fCost и parent = current node.
      * Если соседа нет в open set, добавить его.
   5. Продолжать цикл.
   6. Если open set опустел до достижения цели – пути нет (цель недосягаема).
10. Восстановление пути: когда достигли цели, пройдите от целевого Node по parent-ссылкам до стартового, собирая координаты. Затем инвертируйте список или добавляйте в обратном порядке – это и будет путь от старта к цели.
11. Отладка: проверьте на простых примерах. Например, старт (0,0), цель (5,0) без препятствий: путь должен быть прямая линия. Добавьте препятствие на прямой – путь должен обходить. Попробуйте цель рядом со стартом через стену – убедитесь, что алгоритм обходит стену.
12. Логируйте некоторые моменты, если трудно: например, выводите координаты узла, извлекаемого из open set, его fCost, или количество узлов проверенных. Чтобы не залить консоль, можно ограничиться отдельными тестами.
13. Визуализация поиска. Добавьте в режиме разработки визуальные подсказки, чтобы видеть работу A\*:
14. В OnDrawGizmos() (в редакторе, когда игра на паузе) можно рисовать:
    * Closed set: клетки, которые алгоритм уже посетил – отмечайте их, например, красными полупрозрачными квадратами.
    * Open set: текущие пограничные клетки, которые алгоритм рассматривает – отметьте синими или зелёными квадратиками.
    * Старт/Финиш: можно другим цветом (например, старт – зеленый, финиш – красный).
    * Финальный путь: после поиска, сохраните путь и нарисуйте его – например, используя UnityEngine.LineRenderer для линии или Gizmos.DrawLine между последовательными точками[[62]](file://file_0000000069fc624685d94f067e42a757#:~:text=,%EE%88%80cite%EE%88%82turn3search5%EE%88%82turn3search20%EE%88%81).
15. Для реализации, вы можете вызвать свою функцию FindPath в OnDrawGizmosSelected (так, чтобы каждый раз при выделении объекта в редакторе, прокладывался путь). Однако запускать поиск в Gizmos при большой карте может быть неэффективно. Лучше: запускайте поиск по какому-то триггеру (например, при нажатии клавиши в игре), храните результат и рисуйте его Gizmos до следующего обновления.
16. Визуализация путей мгновенно выявляет ошибки. Например, вы увидите, если алгоритм лезет сквозь стену (значит, где-то walkable неправильно), или если open/closed распространяются странно. Tip: хороший признак правильной работы A\* – фронт волны (open set) распространяется примерно кругом от старта, а закрытые тянутся «пучком» вдоль найденного оптимального пути.
17. Тестирование A\*. Создайте несколько сценариев:
18. Прямой путь: старт и финиш без препятствий. Проверьте, что A\* возвращает прямую (диагональную или прямую) в зависимости от расположения.
19. Обход препятствия: поместите большой блок-стену между стартом и целью. Убедитесь, что путь огибает стену кратчайшим путём (если симметрично, любой из равноценный).
20. Тупик: если цель окружена непроходимыми клетками, алгоритм должен вернуться с пустым путём (или null). Проверьте, что не происходит бесконечного цикла, open set станет пустым и вы обработаете это (например, возвращая пустой список).
21. Проходимость заражения: если вы делали заражённые клетки проходимыми, можно вручную сравнить путь, проходящий через заражённую зону vs обход. Если желаете, попробуйте задать “стоимость” заражённой клетки выше: например, добавляйте +20 к gCost при входе в инфицированную клетку. Тогда A\* будет обходить заражение, если путь сильно удлиняется – полезно для AI, чтобы они обходили опасные зоны. Протестируйте: цель внутри заражённой области – алгоритм пойдёт напрямую (придётся зайти в “красную” зону, иначе не достигнуть), а если цель за заражённой зоной, может обходить вокруг.
22. С помощью профайлера определите, сколько миллисекунд занимает один поиск при разных размерах карты (например, 50x50, 100x100) – на маленьких картах время незначительное, а вот на очень больших (1000x1000) поиск может быть ощутимым. Это подготовит к оптимизациям.
23. Jump Point Search (оптимизация A\*). Реализация JPS значительно ускоряет поиск на больших открытых пространствах, но требует тонкого понимания:
24. Идея: вместо шага от клетки к соседу, JPS позволяет “прыгать” по направлению, пока не потребуется сменить направление. Например, если вы двигаетесь на восток и ничто не мешает, JPS пропустит промежуточные клетки и сразу рассмотрит дальние “принудительные” точки (где появляется препятствие, заставляющее потенциально свернуть). Это сокращает количество рассмотренных узлов.
25. Подход к реализации: вместо простого цикла по всем соседям, в A\* модифицированном под JPS пишут рекурсивную функцию “jump(currentNode, direction)”, которая шагает по сетке в заданном направлении до тех пор, пока:
    * либо достигнет цели,
    * либо найдёт “forced neighbor” (препятствие по диагонали, заставляющее включить боковой сосед),
    * либо упрётся в непроходимое.
    * Каждая такая “принудительная точка” становится узлом, добавляемым в open set.
26. Реализация JPS довольно сложна. Для начального освоения вы можете попробовать реализовать частично:
    * Например, измените обход соседей: вместо добавления всех соседей current, добавляйте только “критические” соседи. В открытом поле JPS должно генерировать гораздо меньше узлов.
    * Найдите в интернете псевдокод JPS (например, на ресурсах, упоминаемых в документе) и попытайтесь встроить его в свой A\*.
    * Проверка JPS: убедитесь, что JPS всегда находит путь той же длины, что и обычный A\*, только быстрее. Сравните длины или набор шагов на нескольких примерах – они должны совпадать[[63]](file://file_0000000069fc624685d94f067e42a757#:~:text=,%EE%88%80cite%EE%88%82turn3search0%EE%88%82turn3search3%EE%88%81). Если где-то JPS даёт иной (больший) путь, значит, ошибка в логике “прыжков” или пропущена “forced neighbor”.
    * Логирование JPS: Для отладки выведите, как JPS “прыгает”. Например: “Jump from (2,2) to (5,2) direction East hit obstacle at (6,2), forced neighbor at (5,3)”. Такие логи помогут отследить, правильно ли определяются контрольные точки. В идеале – минимум узлов, все ключевые повороты на препятствиях.
    * На небольшой карте разница может быть едва заметна по времени, но на большой открытой – JPS должен рассматривать на порядки меньше узлов (например, вместо тысячи – десятки). В профайлере или засечкой времени (Stopwatch) сравните время выполнения A\* vs JPS на большой пустой карте: JPS будет быстрее.
27. JPS – необязательная задача: если чувствуете, что A уже сложен, вы можете пропустить JPS в реализации. Главное – понять идею, прочитать о ней. Для углубления\*: JPS+ и другие улучшения существуют, но выходят за рамки.
28. Иерархический поиск (HPA\*) – концепт. На огромных картах (сотни тысяч клеток) даже A с оптимизациями может быть медленным. HPA решает это, деля карту на регионы:
29. Разбейте сетку на блоки, например 10x10. В каждом блоке выберите несколько “ворот” (portal) – клетки на границах блоков, через которые можно выйти.
30. Постройте граф областей: узлы графа – регионы, ребро между регионами существует, если их порталы смежны и проходимы, вес ребра – расстояние между соответствующими порталами.
31. Когда нужен путь:
    1. Определите, в каких регионах лежат старт и цель.
    2. Найдите путь на графе регионов (абстрактных) – этот поиск проходит через гораздо меньше узлов (например, десятки регионов вместо тысяч клеток).
    3. Затем, для входа и выхода каждого региона – проложите подробный путь внутри региона (на маленькой области) – можно обычным A\*.
    4. Составьте из этих кусков финальный маршрут.
32. HPA требует приличного кодирования (генерация порталов, сохранение промежуточных путей). Полной реализации в рамках квартала, возможно, не потребуется. Но знайте: этот метод даёт резкий\* прирост производительности на больших мирах, особенно если структура карт изменяется нечасто (потому что можно кешировать пути между порталами)[[61]](file://file_00000000e2c461f4bb2cbf5057f4f2ac#:~:text=,%D0%BF%D1%80%D0%BE%D1%85%D0%BE%D0%B4%20%D1%87%D0%B5%D1%80%D0%B5%D0%B7%20%D0%B7%D0%B0%D1%80%D0%B0%D0%B6%D1%91%D0%BD%D0%BD%D1%8B%D0%B5%20%C2%AB%D0%BA%D1%80%D0%B0%D1%81%D0%BD%D1%8B%D0%B5%C2%BB%20%D0%B7%D0%BE%D0%BD%D1%8B).
33. Обновление при изменениях: Если на карте меняются препятствия (строятся здания, вводятся карантины), в HPA не нужно пересчитывать всё с нуля – достаточно инвалидировать\* (пересчитать) те кластеры, которых касается изменение (например, обновить связь порталов только в затронутом регионе)[[64]](file://file_00000000e2c461f4bb2cbf5057f4f2ac#:~:text=%2A%2A%D0%9A%D0%B0%D0%BA%20%D1%83%D1%81%D0%BA%D0%BE%D1%80%D0%B8%D1%82%D1%8C%3A%2A%2A%20,%D0%BF%D0%B0%D1%80%20%C2%AB%D1%80%D0%B5%D0%B3%D0%B8%D0%BE%D0%BD%E2%86%92%D1%80%D0%B5%D0%B3%D0%B8%D0%BE%D0%BD%C2%BB%20%D0%B8%20%D0%BA%D1%80%D0%B0%D1%82%D1%87%D0%B0%D0%B9%D1%88%D0%B8%D1%85%20%C2%AB%D0%BA%D0%BE%D1%80%D0%B8%D0%B4%D0%BE%D1%80%D0%BE%D0%B2%C2%BB). Это намного дешевле, чем пересчитывать глобально.
34. Вы можете не кодировать HPA, но изложите на бумаге/в комментариях, как разбили бы карту и сколько узлов графа получили. Это поможет оценить выигрыш: например, карта 1000x1000 (1,000,000 клеток) разбита на регионы 20x20 = 2500 регионов; поиск на графе из 2500 узлов + локальные A внутри пары десятков регионов вместо поиска по миллиону узлов.
35. Интеграция pathfinding в проект. Теперь, когда есть функция FindPath, воспользуйтесь ей:
36. Юнит-движение: Если у вас есть хотя бы один подвижный объект (например, в Behavior Tree модуля 6 мы предусмотрели “патруль/атака”), можно улучшить его логику: вместо мгновенного “телепорта” к цели, пусть он прокладывает путь и «идёт» по нему.
    * Например, в узле Behavior Tree “MoveToTarget” вызовите path = Pathfinding.FindPath(currentPos, targetPos). Получив список точек, сохраните его. Далее, каждый тик берите следующий шаг из пути и двигайте объект немного в эту сторону. Как только достиг следующей точки, переходите к следующей, и так до конца пути.
    * Отмечайте, если цель или окружение изменились в процессе (например, путь стал недействительным из-за нового препятствия) – тогда проложите путь заново.
    * Это сложнее, поэтому можно сначала имитировать движение: например, сделать задержку Running состояния на N тиков, по окончании считать, что “дошёл”.
37. AI принятие решений: AI может использовать длину пути как фактор. Например, при оценке Utility “поддержать игрока”, можно учесть расстояние между их базами (через pathfinding) – если слишком далеко, приоритет ниже. Или при планировании GOAP: если действие “отправить отряд в точку X” – вы можете заранее оценить время по длине пути.
38. Ограничение запросов: Убедитесь, что не вызываете FindPath слишком часто. Pathfinding – дорогая операция. Если вы, скажем, каждые пару тиков вызываете A для 10 юнитов, это может нагрузить CPU. Оптимизации\*:
    * Кешируйте результаты: например, если десять юнитов идут в одну точку, им можно проложить один путь и копировать.
    * Разносите запросы по времени: 10 юнитов – по 2 юнита в тик (чуть увеличится задержка реакции, но сгладит нагрузку).
    * Используйте асинхронность: Unity Jobs идеально подходят – можно вынести поиск в IJob и выполнять параллельно, если у вас много путей. В этом случае один кадр может запускать несколько pathfinding-job’ов и ждать их окончания к нужному моменту.
    * Если карта статична, рассмотрите предпосчёт: например, вычислить и кешировать кратчайшие расстояния между ключевыми точками (базами, регионами) и использовать их, не гоняя A\* каждый раз.
39. Логирование и отладка
40. Визуализация проверки: С помощью Gizmos убедитесь, что A\* корректен:
41. Нарисованные open/closed узлы должны логически располагаться: closed образуют область от старта до цели по оптимальному пути, open – в основном по периметру этой области (волна). Если видите closed далеко в сторону, не имея отношения к кратчайшему пути, возможно, эвристика неадекватна (не информативна) или есть ошибка, что узлы не исключаются должным образом.
42. Финальный путь – отрисуйте и проверьте, что он действительно самый короткий (можно прикинуть глазом или сравнить с альтернативными маршрутами).
43. Корректность стоимости: Если используете диагонали с затратой 14, проверьте, что путь предпочитает 2 прямых шага (20) над одним диагональным+1 прямым (14+10=24). Если заметите, что предпочитается диагональный путь с чуть большей реальной дистанцией – возможно, ваша эвристика (например, манхэттен) недооценивает и приводит к тому, что алгоритм думает, что диагональ выгоднее. В таком случае скорректируйте эвристику (для 8 направлений лучше h = 10\*max(|dx|,|dy|)).
44. JPS тесты: Обязательно сравните длины путей A и JPS на нескольких случаях[[63]](file://file_0000000069fc624685d94f067e42a757#:~:text=,%EE%88%80cite%EE%88%82turn3search0%EE%88%82turn3search3%EE%88%81). Они должны совпадать до клетки. Если нет – JPS упускает какой-то узел. Логируйте «forced neighbors» и сравните с теоретическим ожиданием: JPS должен останавливать прыжок, когда появляется препятствие по стороне движения или по диагонали (которое в обычном A породило бы альтернативный обход). Отладка JPS может быть нетривиальна, тут поможет внимательное чтение статьи/псевдокода и поэтапный вывод.
45. HPA\* обоснование: Если не реализовали HPA, ничего страшного. Но можно провести мысленный эксперимент: возьмите большую карту и прикиньте, через какие регионы пойдёт путь. Нарисуйте схему на бумаге, выделите порталы. Если видите, что путь пересекает много однотипных открытых клеток – ясно, что A тратит время на них, а HPA перепрыгнул бы регион целиком. Само это понимание помогает проектировать уровни (например, можно заранее планировать “узкие места” – тогда и без HPA AI решения принимает быстрее, потому что вариантов меньше).
46. Производительность: Pathfinding – одна из самых тяжёлых подсистем, особенно когда юнитов или запросов много. Поэтому:
47. Если у вас 100 юнитов, не запускайте всем путь в один кадр – распределите или делайте более грубые цели (неперекалиброванный AI, дергающий pathfinding каждый тик, может съесть всю CPU).
48. Profiler: используйте профиль Unity – запустите сценарий с множеством запросов, посмотрите время функции pathfinding. Оптимизируйте: например, убедитесь, что не создаёте лишних объектов (Node) на каждую клетку – можно переиспользовать или хранить в массиве, сбрасывая значения. Сборщик мусора (GC) не должен работать на каждое построение пути.
49. Отладка эвристики: Heuristic должна не переоценивать реальный путь, иначе A\* теряет оптимальность. Если вдруг вы сделали эвристику, которая иногда больше фактического расстояния – алгоритм мог пропустить оптимальный путь. Обычно манхэттен или диагональная не переоценивают, они допустимы. Если сомневаетесь, тестируйте: любой найденный путь должен иметь длина ≤ h(start); если нашли случай, где найденный путь длиннее эвристики от старта – проблема.
50. Частые баги:
51. Неправильное восстановление пути (например, забыли инвертировать порядок – тогда получается от цели к старту).
52. Ошибки с диагоналями через углы: разрешаете ли движение по диагонали между двумя препятствиями по углам “уголком”? Классически, если оба смежных по горизонтали и вертикали клетки от текущей – непреодолимы, то диагональ тоже должна быть запрещена (чтобы не “просачиваться сквозь угол”). Добавьте эту проверку: перед добавлением диагонального соседа убедитесь, что оба соседних прямых клетки свободны.
53. Границы массива: убедитесь, что не выходите за пределы при проверке соседей (условия 0 <= nx < width, аналогично по y).
54. Сравнение float/интов: лучше всю стоимость в целых (например, 10 и 14), чтобы избегать ошибок накопления и удобнее сравнивать fCost.
55. Не забывайте закрывать узлы: если забыть помещать обработанный узел в closed, алгоритм может зациклиться или повторно его рассматривать. В итоге или зависание, или не оптимально.
56. Не проводите поиск, если старт = цель: можно сразу вернуть путь нулевой длины.
57. Codex для pathfinding: Не стесняйтесь попросить AI помочь:
58. «Объясни, как работает A на сетке 2D»\* – для уверенности.
59. «Почему алгоритм A не находит путь, хотя должен»\* – предоставьте свой код, AI может найти упущенное условие или баг.
60. «Как реализовать Jump Point Search» – он может дать упрощённый псевдокод, по которому скорректировать свой.
61. «Как улучшить производительность поиска пути» – может напомнить об асинхронности, кешировании, что мы уже обсудили.
62. Воспользуйтесь Codex, чтобы написать, например, приоритетную очередь (binary heap) для open set: запрос «Реализуй бинарную кучу на C#» – и вставьте полученное, так поиск ускорится (List.Sort каждый раз – лишние затраты).
63. С помощью AI протестируйте критические функции: попросите сгенерировать юнит-тест для FindPath на известной сетке (вы можете захардкодить небольшую карту в массив и вызвать функцию, проверить ожидаемый путь). Это убедит, что функция работает правильно в автономном режиме.

## Модуль 8: Оптимизация производительности и масштабирование

### Краткое описание модуля

К этому моменту у нас множество систем: симуляция инфекции, экономика, AI, pathfinding и т.д. В небольших масштабах (десятки агентов, карта 100x100) они могут работать на MonoBehaviour без проблем. Но если игра разрастётся (сотни агентов, карта тысячники, большой ИИ), нам нужно подготовиться к оптимизации. В этом модуле мы разберём стратегии масштабирования: - Сначала профилируем и выявляем “горячие точки”. - Затем оптимизируем самые тяжёлые циклы, перенося их на многопоточные Jobs с Burst-компиляцией (нативный код) – это даст ускорение на многоядерных CPU. - Если и этого мало, посмотрим в сторону DOTS/ECS (Entity Component System) – кардинальная переделка под дата-ориентированную архитектуру, но позволяющая симулировать тысячи объектов с фиксированным шагом крайне эффективно. - Дополнительно, улучшим производительность рендеринга и загрузки: используем пуллинг объектов для частых созданий/уничтожений, настроим Tilemap для минимизации draw calls, и внедрим Addressables для управления памятью ассетов. - Цель: Научиться пошагово усиливать производительность по мере необходимости, не делая преждевременных оптимизаций, но и зная заранее узкие места архитектуры.

### Цели обучения

* Понять, как использовать профайлер Unity и Frame Debugger для измерения производительности CPU, сборщика мусора (GC) и рендеринга. Уметь находить функции, которые потребляют много времени, и понимать, почему (например, большой цикл, частое выделение памяти, слишком много вызовов Update).
* Научиться переписывать вычислительно тяжёлый код под Jobs System Unity: разбивать работу на небольшие параллельные задачи (job’ы), использовать NativeArray для данных, применять [BurstCompile] для максимального ускорения. Понять ограничения (нельзя обращаться к Unity Object внутри job, нельзя выделять память в job, нужны безопасные контейнеры).
* Освоить базовые приёмы отладки job’ов: запускать без Burst (тогда можно ставить брейкпоинты), использовать Burst Debug режим.
* Понять, когда стоит перейти на ECS: при очень большом количестве однотипных объектов, где ООП-подход с MonoBehaviour начинает тормозить. Получить общее представление о компонентах, системах и группах ECS, о фиксированной группе симуляции и буферах команд (EntityCommandBuffer) для изменений.
* Научиться профилировать память: отслеживать утечки (например, неосвобождённые NativeArray), контролировать использование Addressables (правильно ли выгружаются ассеты, нет ли дублирования в памяти).
* Внедрить Object Pooling для часто создаваемых объектов (например, визуальные эффекты, пули), чтобы избежать лишних затрат на Instantiate/Destroy.
* Научиться эффективно использовать Codex для оптимизационных подсказок: просить проверить, где узкое место, или переписать метод с учетом многопоточности.

### Основные задачи (по возрастанию сложности)

1. Профилирование базовой игры. Прежде чем что-либо оптимизировать, измерьте:
2. Запустите игру в наиболее “нагруженном” сценарии, который у вас сейчас есть. Например: максимальная площадь заражения, много построек, AI активно строит, юнит что-то патрулирует, несколько запросов pathfinding. Откройте Profiler (Window → Analysis → Profiler) и Frame Debugger.
3. В профайлере обратите внимание на:
   * Вкладку CPU Usage: какие методы/frame пожирают больше всего времени. Например, может оказаться, что InfectionSystem.Tick занимает 40% кадра (потому что там двойной цикл по большой сетке), или Pathfinding.FindPath скачет при вызове и отъедает 10 мс. Выпишите топ-3 самых тяжёлых места.
   * GC Alloc: посмотрите, генерируется ли сборка мусора каждый кадр (желтые всплески). Если да – выясните, что создаётся. Часто причины: создание новых объектов/коллекций каждый Update. Например, if FindPath создает много new Node, их потом собирает GC. Или EventBus генерирует что-то. Зафиксируйте объем (в байтах). Задача – минимизировать частые аллокации, чтобы GC не мешал (идеально 0B per frame в стабильном состоянии).
   * Timeline: переключитесь в режим Timeline, рассмотрите кадр. Видно ли, что какие-то системы можно выполнять параллельно (например, Infection и AI могли бы идти вразнобой)? Unity по умолчанию выполняет весь ваш код в одном потоке (Main Thread). Нам нужно выявить, какие куски можно вынести в job’ы.
4. В Frame Debugger посмотрите рендер: сколько Draw Calls и SetPass. Возможно, Tilemap разбивает на чанки (и хорошо), но если вы использовали Tilemap с Individual mode, draw calls могут быть сотни. Отмечайте: хотим <100 draw calls в идеале для 2D, иначе GPU перегружается.
5. Результат этого шага – понимание: что тормозит прямо сейчас? Например: “Система инфекции – 5 мс, Behaviour Tree – 2 мс, Pathfinding (при запросе) – 3 мс, отрисовка – 4 мс, GCAlloc – 0.5 KB/frame”. С такими данными можно решать, что оптимизировать.
6. Вынос тяжёлых циклов в Jobs + Burst. На основе профайлера выберите самую затратную функцию – скорее всего, обновление инфекции (двойной цикл по всей сетке) или массовое вычисление influence-карт, или обработка сотен агентов. Попробуем оптимизировать на примере InfectionSystem:
7. Подготовка данных: переводим нужные структуры в формат, пригодный для job. Например, у нас CellInfection[,] grid. В Unity Jobs нельзя directly использовать многомерный массив или List, но можно NativeArray<CellInfection> одномерный. Представим сетку как одномерный массив длины widthheight (с индексом idx = ywidth + x). Создайте NativeArray<CellInfection> gridNative и скопируйте данные туда. Либо создайте отдельный NativeArray<float> infectionLevels и NativeArray<int> infectionStates (если struct сложен).
8. Определение Job: Напишите struct InfectionJob : IJobParallelFor (если хотите параллельно по индексу) или IJob (если будете внутри делать свой параллелизм). Например:

* [BurstCompile]  
  struct InfectionJob : IJobParallelFor {  
   public NativeArray<CellInfection> current;   
   public NativeArray<CellInfection> next;  
   public int width;  
   public int height;  
   public void Execute(int index) {  
   int x = index % width;  
   int y = index / width;  
   // ... вычислить новый уровень и состояние на next[index] на основе current[...] соседей  
   }  
  }
  + Используйте подход с двумя массивами (текущий/следующий) как раньше. Можно даже оптимизировать: хранить только level отдельно, а состояние S/I/R хранить как byte для компактности.
  + Внутри Execute нельзя обращаться к обычным полям/свойствам Unity или к managed-объектам. Поэтому все данные надо передать как простые значения или NativeArray. Также нельзя использовать List или Dictionary. Поэтому, если вам нужны, скажем, массив соседних индексов – можете вычислять на лету или заранее передать массив смещений соседей.
  + BurstCompile: Атрибут [BurstCompile] перед классом job – заставит компилировать его в высокопроизводительный код, но есть ограничения: нельзя выводить в лог внутри bursted job (Debug.Log не поддерживается), нельзя использовать managed memory.

1. Запуск Job: В SimulationLoop.Update, когда приходит время обновить инфекцию:
   * Перед циклом while или где удобно, можно заполнить gridNative текущими данными. (Если gridNative у вас Persistent, вы можете его обновлять в OnEnable, а потом только перезаписывать изменённые значения).
   * Создайте экземпляр InfectionJob job = new InfectionJob { current = gridNative, next = gridNextNative, width = w, height = h };
   * Получите JobHandle handle = job.Schedule(width\*height, 64); – это запустит параллельно на нескольких потоках, разбивая на группы по 64 индекса (подбирайте batchSize, 64 – начальный выбор).
   * Синхронизация: вам нужно дождаться конца работы job прежде, чем использовать результат. В простом случае делаем handle.Complete(); – это блокирует главный поток, пока job не закончит. По сути, это аналог последовательного выполнения, но job выполняется параллельно, и Complete дождётся её (к этому времени часть Main Thread могла выполнять другие вещи, но в нашем случае после запуска infectionJob нам, возможно, нечего делать – но представьте, мы могли параллельно и AI считать).
   * После Complete(), данные в gridNextNative заполнены. Скопируйте их обратно в обычную структуру (или, если вы полностью переходите на NativeArray как основное хранилище, можно хранить так). Например, можно иметь float[] infectionLevel для UI или отладки, или сразу перебирать NativeArray при визуализации.
   * В конце, не забудьте: если NativeArray создан с Allocator.TempJob или Allocator.Persistent, нужно его .Dispose() когда он не нужен. Хорошо, если gridNative вы сделали полем и держите постоянно (Allocator.Persistent) – тогда нет аллокации каждый кадр, и Dispose сделаете в OnDestroy.
2. Проверка результата: Убедитесь, что переход на job ничего не сломал. Лучший способ – оставить старую реализацию (MonoBehaviour цикл) и новую (job) и сравнить результаты на одном кадре:
   * Например, на 10-м тик делайте и старый Tick, и новый (новый – не применяя к основному состоянию), и сравните nextGridOld vs gridNextNative. Если различия – есть баг в job.
   * Если все ок, вы можете удалить старый код или оставить комментированным.
3. Измерение эффекта: Запустите профайлер на том же сценарии. В Timeline вы теперь увидите, что на Main Thread выполнение InfectionJob занимает почти ноль (только подготовка), а основная работа происходит в Worker Threads параллельно. В идеале, например, вместо 5 мс на главном потоке, вы получите <1 мс на главном и, скажем, 2 мс на фоновых (которое не блокирует кадр, если у вас многоядерный CPU). Это ускорение. Если у вас 4 ядра, теоретически 5 мс могут превратиться ~в 1.3 мс (5/4), плюс накладные расходы.
4. Частые ошибки при Jobs:
   * Забыли Complete() – тогда игра пойдёт дальше, и вы начнёте использовать gridNextNative до того, как job завершился, что плохо (может быть исключение или неправильные данные). Можно вместо ручного Complete использовать механизм JobHandle: например, цепочку job’ов илиDependency injection, но это сложно – пока просто Complete.
   * Неправильный Allocator: TempJob нужно освобождать через .Dispose() вручную или он автоматически освободится через короткое время, но лучше явно.
   * Race conditions: если бы у нас, например, два job – нужно учитывать, не конфликтуют ли они. В Unity Jobs, если один job пишет, а другой читает один массив – надо указывать job.WithReadOnly(array) или разделять на Write/ReadDependency, но в простейшем случае, когда job один, это не нужно.
   * Обращение вне массива: в параллельном коде это особенно опасно – будет сложно отладить. Убедитесь, что индексы верны. Добавьте защиту: например, проверка if (x < width-1) вокруг доступа к правому соседу и т.п.
   * Debugging: Вы не можете просто Debug.Log внутри job (под Burst он игнорируется). В Unity Burst компиляторе есть опция Native Debug Mode, её можно включить в Burst меню – тогда job выполняется без оптимизаций, и можно хотя бы try-catch. Однако, чаще проще отлаживать job, запустив его без Burst (временно убрать [BurstCompile]) – тогда можно даже Debug.Log (медленно, но для теста) или attach-дебагер. После отладки вернуть [BurstCompile].
5. Расширение: По аналогии с InfectionSystem, можно вынести и другие подсистемы:
   * Influence Map: если вы вычисляете поле влияния врага/обороны каждый тик полным сканированием карты, сделайте job, параллельный по частям сетки.
   * Pathfinding: можно выполнение одного поиска пути сделать job’ом, но больше выигрыш, если вы запускаете сразу пачку поисков. Например, 10 запросов – можно запустить 10 job’ов параллельно (или 1 IJobParallelFor, который в Execute(index) запускает маленький поиск на подкарте – но pathfinding не тривиально параллелится на уровне одного запроса). Unity предлагает NavMesh за нас, но если пишете свой – держите в голове.
   * AI Utility: если Utility AI оценивает 5 целей, а ещё на 5 регионов – 25 оценок – можно распараллелить, но это мелочь. GOAP планирование – тоже скорее sequential (не стоит параллельно).
   * Массовая физика или столкновения: у нас нет явно, но если бы были сотни пуль – их обновление/проверка столкновений – кандидат для jobs.
6. Введение в ECS (Entities). Перенос всего проекта на ECS – дело трудоёмкое, но можно попробовать Proof-of-Concept на одной системе, например, на той же инфекции:
7. Entities Setup: Подключите пакет Entities (если ещё нет, через Package Manager). Создайте компонент:

* struct InfectionComponent : IComponentData {  
   public float level;  
   public byte state;  
  }
  + Под Entities ваша сетка превратится в множество сущностей – по одной на клетку. Это не всегда оптимально (1млн сущностей – тоже нагрузка), но ради примера.
  + В OnCreate какого-то системного классa (например, InfectionSystem : SystemBase) – создайте все сущности:
  + Entities.ForEach(...) // либо с EntityManager вручную
  + Например,
  + for(int y=0; y<height; y++){  
     for(int x=0; x<width; x++){  
     Entity e = entityManager.CreateEntity();  
     entityManager.AddComponentData(e, new InfectionComponent { level = initialLevel, state = 0 });  
     // Можно добавить компонент Position (или использовать Entity's index as position mapping)  
     }  
    }
  + Это разовое создание в начале (дорого, но раз).

1. Infection System (ECS): Напишите системный класс:

* [UpdateInGroup(typeof(FixedStepSimulationSystemGroup))]  
  public partial class InfectionUpdateSystem : SystemBase {  
   protected override void OnUpdate() {  
   float dt = ...; // получить deltaTime (в ECS fixed dt можно через Time.DeltaTime, если настроено)  
   // Параллельный job:  
   Entities  
   .WithAll<InfectionComponent>()  
   .ForEach((ref InfectionComponent inf) => {  
   // простейшее: если state=1 (Infected) уменьшаем level  
   if(inf.state == 1) { inf.level -= 0.1f \* dt; if(inf.level <= 0) inf.state = 2; }  
   // заражение соседей – сложнее: ECS хорош, когда можно в один проход  
   // Здесь в ECS можно использовать 8 отдельных System, или 8-соседний подход через stencils.  
   }).ScheduleParallel();  
   }  
  }
  + Мы пометили [UpdateInGroup(FixedStepSimulationSystemGroup)] – значит, система будет работать с фиксированным шагом (например, 10 тиков/сек), аналогично нашему SimulationLoop[[65]](file://file_0000000069fc624685d94f067e42a757#:~:text=3%29%20DOTS%2FEntities%20%E2%80%94%20%D0%A4%D0%B8%D0%BA%D1%81,%EE%88%80cite%EE%88%82turn0search2%EE%88%82turn1search19%EE%88%81). Unity сама позаботится, чтобы вызывался нужный Group с заданной периодичностью (можно настроить).
  + Здесь Entities.ForEach – уже распараллеливает по сущностям. С Burst по умолчанию. Очень удобно, если работа локальная, но у нас проблема – нужно учитывать соседей. Как в ECS узнать соседние клетки?
  + Либо через доп. компонент Position (тогда можно в ForEach внутри делать look-up соседей через ComponentDataFromEntity – но это плохо, random access, медленно).
  + Либо, более ECS-way: сделать отдельную систему, которая, например, проходится по всем сущностям и если у сущности state=1, повышает уровень у соседей (в ECS можно хранить сетку как NativeArray и передавать в job).
  + ECS не панацея для клеточных автоматов, unless вы пользуетесь их Grid/Physics/GridQuery API.
  + Выигрыш ECS: В чём же профит? ECS + Jobs + Burst дает то же, что мы сделали вручную: параллельный код на C, но упакованный удобно. Преимущество в том, что Entities могут динамически создаваться, и ECS позаботится, чтобы это происходило безопасно – через EntityCommandBuffer (ECB).
  + Например, если при достижении какого-то условия мы хотим создать новую сущность/объект, мы не можем прямо внутри job вызвать CreateEntity (это structural change). Нужно записать команду ecb.CreateEntity() и система исполняет её после завершения всех job данного этапа (тем самым избегая конфликтов)[[65]](file://file_0000000069fc624685d94f067e42a757#:~:text=3%29%20DOTS%2FEntities%20%E2%80%94%20%D0%A4%D0%B8%D0%BA%D1%81,%EE%88%80cite%EE%88%82turn0search2%EE%88%82turn1search19%EE%88%81).
  + Для нас, в инфекции, не особо нужно создавать/удалять сущности, только изменять данные (это разрешено напрямую).
  + Когда переходить на ECS: Если ваш проект действительно требует симуляции тысяч агентов с физикой, рендерингом – ECS даст выигрыш. Но он потребует многому научиться (Data-Oriented Design, писать систему вместо методов). В рамках квартала вы, возможно, только чуть-чуть коснётесь ECS. Поэтому:
  + POC: Реализуйте маленький кусочек (например, диффузию инфекции) и убедитесь, что оно работает, сравнив с основной системой. Запустите профайлер: ECS-система будет видна на вкладке Entities и будет тоже раскидываться на воркеры.
  + Правила ECS: Помните – все структурные изменения через ECB (иначе sync point и потеря преимуществ)[[65]](file://file_0000000069fc624685d94f067e42a757#:~:text=3%29%20DOTS%2FEntities%20%E2%80%94%20%D0%A4%D0%B8%D0%BA%D1%81,%EE%88%80cite%EE%88%82turn0search2%EE%88%82turn1search19%EE%88%81), фиксированный шаг через группу (как сделали), и не смешивайте MonoBehaviour логику с ECS на те же данные одновременно (можно, но аккуратно).
  + Отладка ECS: Отслеживайте в Entities debugger (Window → Entities). Убедитесь, что ваша система в правильной группе (FixedStepSimulationSystemGroup) и что fixedDeltaTime соответствует SimulationLoop tick (можно настроить в World Time).
  + Вывод: Попробовав ECS, решите, нужно ли внедрять его во всё. Возможно, вы ограничитесь job-ами на критичных местах, не переписывая всё на ECS – это нормально для малого проекта. Помните: “сначала сделать, чтобы работало; потом – оптимизировать”. У вас уже есть рабочие MonoBehaviour-системы, и вы точечно их ускоряете job’ами. Переписывать всё на ECS имеет смысл, если планируется действительно массовая масштабируемость (например, тысячи юнитов) или если вы изначально хотели освоить DOTS как цель.

1. Другие оптимизации. Помимо CPU-вычислений, есть аспекты GPU и памяти:
2. Объединение отрисовки: Проверьте настройки Tilemap. Если у вас несколько Tilemap-слоёв, старайтесь, чтобы большинство были в режиме Chunk – тогда Unity рисует их по блокам и меньше вызовов. Если есть особые слои (где нужны индивидуальные спрайты между, как указано в документе), используйте Sprite Atlas для них, чтобы Unity мог батчить и их[[66]](file://file_00000000e2c461f4bb2cbf5057f4f2ac#:~:text=%D0%9A%D0%BE%D1%80%D0%BE%D1%82%D0%BA%D0%B8%D0%B5%20%D0%B4%D0%BE%D0%BF,%D0%B8%D0%BD%D0%B0%D1%87%D0%B5%20%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D1%8C%20%D1%80%D0%B0%D1%81%D0%BF%D1%83%D1%85%D0%B0%D0%B5%D1%82). Откройте Frame Debugger: он покажет, где батчинг прерывается (например, разные материалы – значит, атлас не используется, или разные слои – не избежать).
3. Object Pooling: Если у вас появляются объекты во время игры (например, вы планировали снаряды, визуальные эффекты или даже те же башни строятся и уничтожаются часто), вместо Instantiate/Destroy каждый раз – сделайте пул:
   * Например, пул List<GameObject> предварительно созданных снарядов (неактивных). При выстреле – берёте из пула (SetActive true, перемещаете), при попадании – отключаете и возвращаете в пул. Unity даже имеет ObjectPool<T> класс под эту задачу[[67]](file://file_00000000e2c461f4bb2cbf5057f4f2ac#:~:text=%E2%80%94%20Utility,T%3E%60%29%20%EE%88%80cite%EE%88%82turn6search6%EE%88%81). Это резко снижает нагрузку на GC и CPU при массовом появлении/удалении объектов.
   * В вашем проекте, пул могли бы требовать разве что визуальные части (например, мигающие индикаторы, если бы вы ими спамили). Но знать стоит, т.к. RTS могут иметь эффекты (взрывы, пули).
4. Addressables и память: Если вы загружаете ресурсы (например, решите подгружать спрайты, аудио) – используйте Addressables. Главное – соблюдайте пару правил:
   * Загруженное -> освобожденное: каждая пара Addressables.LoadAsync -> Addressables.Release, чтобы не копились невидимые объекты в памяти[[27]](file://file_00000000e2c461f4bb2cbf5057f4f2ac#:~:text=,%D0%BF%D0%BE%20%D1%81%D0%BF%D0%BE%D1%81%D0%BE%D0%B1%D0%B0%D0%BC%20%D0%B7%D0%B0%D0%B3%D1%80%D1%83%D0%B7%D0%BA%D0%B8%2F%D1%87%D0%B0%D1%81%D1%82%D0%BE%D1%82%D0%B5%20%D0%B8%D1%81%D0%BF%D0%BE%D0%BB%D1%8C%D0%B7%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F%20%EE%88%80cite%EE%88%82turn2search10%EE%88%82turn2search16%EE%88%81).
   * Группируйте адресуемые ассеты по тому, как будут использоваться (например, весь UI в одном груп, все большие спрайты в другом), так вы сможете подгружать/выгружать целыми группами.
   * Проверьте Memory Profiler (есть отдельный пакет) или хотя бы профайлер вкладку Memory: посмотрите, нет ли всплеска памяти после, например, сцены загрузки/выгрузки. Addressables часто ловят на том, что загрузили, а Release забыли – тогда на каждый повтор загрузка дублируется в памяти.
5. Расчёты AI: Если у вас сложный Utility или планер, убедитесь, что он не срабатывает слишком часто:
   * Стратегический AI мы и так сделали раз в N тиков. Можно даже реже: например, цели пересматриваются раз в секунду, этого хватит.
   * Tactical AI/Behavior Tree – работает каждый тик, но там дешёвые операции. Однако, если бы BT стал сложнее, тоже не стесняйтесь пореже делать тяжелые проверки.
   * Разделение вычислений: когда невозможно распараллелить, попробуйте разнести по кадрам. Например, у вас 50 юнитов и сложный расчет для каждого – не делайте всех в один кадр. Делайте 10 юнитов в кадр (каждому обновление раз в 5 кадров). Для стратегических решений это вообще нормально.
6. Планировщик задач: Можно внедрить простой менеджер, который будет распределять задачи по кадрам (например, у вас очередь pathfinding запросов, он берет по одному-два на кадр). Это несложно – написать обёртку над FindPath, которая либо сразу выполняет (если мало запросов), либо ставит их в очередь, а Update будет поштучно вызывать. Так вы избежите спайков.
7. Кэширование результатов: Если один и тот же расчет повторяется с теми же входными данными – закэшируйте. Например, AI каждый тик запрашивает “сколько заражено клеток?” – посчитайте это в InfectionSystem один раз за тик и сохраните, а AI пусть берет готовое значение. Такие мелочи устраняют дублирование труда.
   * Особенно актуально для GOAP: если несколько действий используют одно условие, лучше подсчитать его один раз и хранить.
8. Выключение неиспользуемого: Unity тратит время на каждый активный Update(). Если у вас есть объекты/скрипты, которые сейчас ничего не делают – выключайте их (enabled=false) или уничтожайте, чтобы не занимали цикл.
   * Пример: если у вас будет много снарядов, не делайте для каждого Update – лучше пусть PoolManager дергает их движение.
   * Сейчас, в нашем проекте, вероятно, каждый скрипт выполняется, но их не так много – это не узкое место. Но на будущее: ограничить количество пустых Update.
9. Логирование и отладка
10. Profiler Iteration: Повторяйте процесс профилирования на каждом шаге оптимизации:
11. Сохраняйте снимки профайлера (есть функция Save Profiler Data). Например, baseline и после внедрения job’ов.
12. Сравнивайте: снизилось ли время на функцию X? Переместилась ли нагрузка на Worker Threads? Нет ли новых подозрительных всплесков (например, After job – GCAlloc уменьшился, но вдруг где-то стало больше, значит, возможно, мы создаём NativeArray каждый кадр и не освобождаем).
13. GPU: Если были проблемы с количеством draw calls, после оптимизации Tilemap/атласа – стало ли меньше SetPass? Если да, FPS вырастет, если GPU была узким местом.
14. Следите за Main Thread frame time – для 60 FPS он должен быть ~16 ms или ниже. Если у вас ранее было 30 ms, а после оптимизации стало 12 ms – цель достигнута.
15. Ошибки Race Condition: При параллелизме легко не заметить ошибку, которая проявляется редко (например, два job записывают в одну переменную одновременно). Если сталкиваетесь со странными багами (нестабильные результаты) – возможно, гонка данных.
16. Для отладки таких: временно отключите Burst и выполните job в однопоточном режиме, сравните результаты. Используйте Conditionals/Asserts: например, если два job считают сумму, можно после всех job сверить с однопоточным вычислением.
17. Unity’s NativeArray имеют функцию .IsCreated – проверяйте и выбрасывайте предупреждение, если вдруг массив уже Disposed (значит, где-то двойное освобождение).
18. Если работа с EntityCommandBuffer – убедитесь, что все создания/удаления сущностей делаются через него, а не напрямую EntityManager внутри job – иначе Unity выдаст исключение (ECS не позволит, кстати).
19. Частые подводные камни:
20. Неосвобождённые NativeArray – утечка памяти. Unity Editor предупредит в консоли (кажется, о TempJob утечках), но не всегда. Следите вручную. Если массив нужен постоянно, делайте Persistent и храните, переиспользуя.
21. Burst не поддерживает обращение к классам/ссылкам – если очень надо, пометьте поле [NativeDisableParallelForRestriction] или [NativeDisableUnsafePtrRestriction] – но лучше перепроектировать. Обычно, если нужно что-то сложное (например, поиск пути) – оставьте на Main Thread или делайте отдельной одиночной job, а не параллельно клеткам.
22. Попытка “параллелить то, что не параллелится” – иногда overhead job’ов перевесит выгоду. Например, если у вас всего 10 объектов и простой код – job может только усложнить. Всегда сопоставляйте цену разбивки: job-менеджер тоже тратит время на запуск. Поэтому оптимизируйте действительно большие задачи, а не всё подряд.
23. Валидация: после оптимизации обязательно прогоните все игровые тесты, которые у вас были. Производительность – хорошо, но не ценой сломанной логики.
24. Преждевременная оптимизация – зло: мы оптимизируем то, что профайлер показал существенным. Не тратьте время на job там, где и так <1% кадра. Лучше сосредоточиться на том, что было узким местом.
25. Применение Codex:
26. «Как преобразовать этот метод в Job?» – можете показать Codex кусок кода, он подскажет, как вынести.
27. «Пример использования IJobParallelFor в Unity» – получив пример, сопоставьте со своей задачей.
28. «Почему Burst не работает с моим кодом?» – если получили ошибку компиляции, AI может подсказать, что не так (например, использован недопустимый тип).
29. «Как работает EntityCommandBuffer в Unity ECS?» – Codex может объяснить или дать шаблон, как правильно создавать сущности внутри системы.
30. «Как правильно профилировать Unity игру» – AI может перечислить шаги, вдруг вы что-то забыли (например, отключить Deep Profile, если он не нужен, или включить Profile Editor).
31. «Какие ещё способы оптимизации RTS?» – AI может упомянуть, например, fixed-point math вместо float (редко нужно), или специализированные структуры. Это даст идеи на будущее.
32. Используйте AI для ревью: пусть посмотрит ваш оптимизированный код – «Не пропустил ли я Dispose где-нибудь?» – он часто может заметить, например, отсутствие Dispose.
33. AI-друг поможет и с Addressables: «Как выгрузить Addressable после использования?» – подскажет про Addressables.Release и тонкости.
34. Помните, что Codex – не панацея: оптимизация требует измерения, он может только общие вещи сказать. Но свериться с советами – полезно.

### Перед переходом к итогам:

После проведённых оптимизаций, у вас игра должна: - Работать плавно в ожидаемых масштабах (проверено, например, 200 тиков на средней карте в 60 FPS держится). - Не быть замеченных утечек памяти (профайлер память стабилен, утечки NativeArray отсутствуют). - Критичные подсистемы (инфекция, массовые вычисления) – вынесены на job’ы, их вклад в загрузку кадра снизился. - Вы понимать, как дальше масштабировать: например, если завтра нужно добавить 1000 агентов, вы знаете, что их логику тоже можно распараллелить или перевести в ECS. - В процессе вы сохранили корректность (юнит-тесты и игровые тесты проходят). - Итог: Вы увидели, что архитектура, заложенная ранее (фиксированный тик, разделённые системы) очень способствует оптимизации: мы могли независимо ускорять InfectionSystem, AI и т.д., не переписывая всё. Это и есть преимущество грамотной архитектуры.

## Общие советы: эффективная работа с Codex и Visual Studio Code

Чтобы максимально использовать возможности AI-помощника (Codex GPT) и среды VS Code, вот несколько рекомендаций и приёмов:

* Установка и настройка: Убедитесь, что расширение Codex (например, CodeGPT) установленно и авторизовано. Часто требуется ввести API-ключ OpenAI или залогиниться. В VS Code обычно появляется боковая панель чата или команды в Command Palette.
* Обращение к Codex через чат: В VS Code, скорее всего, есть команда открыть чат CodeGPT. Вы можете задавать вопросы на любом языке (AI понимает и русский, и английский; технические термины лучше оставлять по-английски, чтобы избежать недопонимания).
* Контекст и файлы: Используйте возможность давать контекст. Например, расширение CodeGPT поддерживает упоминание файла или выделенного кода. Вы можете выделить фрагмент кода и вызвать команду Ask CodeGPT – AI увидит только этот фрагмент. Или в чате написать @MyClass.cs – тогда ассистент учтёт содержимое файла MyClass.cs[[68]](https://marketplace.visualstudio.com/items?itemName=DanielSanMedium.dscodegpt#:~:text=Chat%20Commands). Это очень полезно, когда вопрос касается конкретного кода – не нужно копировать вручную.
* Команды помощника: Многие расширения предлагают готовые команды (в CodeGPT чат-агенте используются префиксы):
* /Fix – автоматически найти и предложить исправление ошибок в выделенном коде[[68]](https://marketplace.visualstudio.com/items?itemName=DanielSanMedium.dscodegpt#:~:text=Chat%20Commands). Если у вас ошибка компиляции или логическая, можно выделить проблемный метод и набрать /Fix: AI попытается переписать код правильно. Пример: вылетает NullReferenceException – AI может заметить, что забыли проверку на null или перепутали порядок инициализации.
* /Refactor – улучшить код: привести к лучшему стилю, разбить на функции, убрать дублирование[[69]](https://marketplace.visualstudio.com/items?itemName=DanielSanMedium.dscodegpt#:~:text=,context%20for%20the%20AI%20agent). Полезно, если код работает, но выглядит громоздко – AI предложит более чистую версию. Всегда проверяйте результат, но зачастую он дельный.
* /Document – сгенерировать комментарии или описание для выделенного кода[[68]](https://marketplace.visualstudio.com/items?itemName=DanielSanMedium.dscodegpt#:~:text=Chat%20Commands). Можно использовать, чтобы получить понятное объяснение функции. Это и для себя полезно (лучше поймёте, что написали), и для практики написания doc-комментариев.
* /Unit Testing – сгенерировать шаблон юнит-тестов для выделенного кода[[69]](https://marketplace.visualstudio.com/items?itemName=DanielSanMedium.dscodegpt#:~:text=,context%20for%20the%20AI%20agent). Помощник может сразу придумать 2-3 тест-кейса. Вы можете их взять и доработать под свои нужды. Это ускоряет создание тестов.
* Команды могут отличаться в других расширениях, но идея одна: они экономят время на рутинных действиях.
* Примеры эффективных запросов:
* «Как мне оптимизировать обновление 1000 объектов в Unity?» – AI может предложить использовать Jobs или ECS, как мы обсуждали.
* «Почему метод X выдаёт ошибку Y?» – укажите конкретную ошибку/стек-трейс. Часто Codex сможет объяснить: например, NullReferenceException – подскажет, что возможно объект не присвоен в инспекторе или не инициализирован.
* «Не понимаю, как работает этот код» – выделите кусок кода, используйте /Document или спросите: «Расскажи пошагово, что делает этот метод». AI опишет своими словами. Это очень помогает, когда разбираетесь с чужим или старым кодом.
* «Приведи пример использования класса X из Unity API» – Codex может сгенерировать небольшой пример. Например: «Как использовать NavMeshAgent для перемещения?» – выдаст код с agent.SetDestination(). Проверяйте с документацией, но часто примеры корректны.
* «Составь запрос LINQ для фильтрации списка объектов по полю» – AI справляется с такими задачами, сэкономив вам время вспоминания синтаксиса.
* «Создай псевдокод для функции, выполняющей ...» – иногда полезно, прежде чем писать код, попросить AI набросать структуру. Вы получите план прямо в коде.
* Интеграция с VS Code при написании кода:
* Если у вас подключен GitHub Copilot или аналог авто-дополнения, он может дописывать строки, когда вы пишете комментарий или начало функции. Попробуйте: введите // функция, которая вычисляет фиксированный тик... (на русском Copilot тоже иногда реагирует) и несколько пустых строк – возможно, авто-дополнение предложит реализацию SimulationLoop. Такие инструменты работают на основе AI (Codex), только в инлайн-режиме.
* CodeGPT тоже может подсказывать, но чаще через чат. Вы можете сочетать: Copilot для автодополнения, CodeGPT для вопросов/отладки. Многие разработчики так и делают.
* VS Code полезные фишки:
* Автопереформатирование: нажмите Shift+Alt+F (или Configure Format Document) – код выровняется по стилю. Если подключен .editorconfig или Unity C# Extension, будет по стандарту Unity. Красивый код легче читать (и AI тоже).
* Мульти-курсор: Alt+клик в нескольких местах – можно писать одно и то же сразу в нескольких строках. Удобно, если нужно проставить что-то однотипное в ряде полей.
* Поиск по проекту: Ctrl+Shift+F – найти все упоминания, например, "TODO". Вы так не забудете части кода, которые помечали для доработки.
* Интеграция Unity: Установите расширение Unity Debugger для VS Code, если еще нет. Оно позволит ставить breakpoints и запускать игру из VS Code с остановкой в них. Запустите Unity в режиме debug (Assets → Open C# Project, затем в VS Code выберите attach to Unity Editor). Теперь, если, скажем, InfectionSystem работает не так – поставьте точку останова внутри Tick и запустите сцену. Игра приостановится на этой строке, а вы можете исследовать значения переменных в VS Code. Это незаменимо для сложной отладки.
* Unity-specific Snippets: Есть расширения, добавляющие сниппеты. Например, набираете for и два раза Tab – появляется шаблон цикла. Или prop – шаблон свойства. Используйте их, чтобы не писать шаблонный код вручную. AI тоже умеет делать сниппеты, но встроенные быстрее.
* Лучшие практики использования AI:
* Конкретизируйте запрос. Вместо: “помоги, не работает” – укажите, что ожидалось и что получилось. Например: “В функции InfectionSystem.Tick ожидал, что все Susceptible с уровнем >0.5 перейдут в Infected, но этого не происходит. Вот код... Что не так?” – такой подробный вопрос заставит AI анализировать именно вашу проблему.
* Проверьте несколько источников. Codex – не единственный ресурс. Изучайте официальную документацию Unity, блоги, форумы. AI иногда может ошибаться или устаревшую информацию дать (например, про старый API). Всегда сверяйте ответы с актуальной документацией.
* Не доверяйте слепо. Используйте AI как помощника, но ответственность на вас. Обязательно тестируйте код, сгенерированный AI, на простых примерах. Он ускоряет рутину, но может не знать всех нюансов вашего проекта.
* Учитесь у AI. Если Codex предложил оптимизированное решение, постарайтесь понять, почему оно лучше. Так вы растёте как разработчик. Например, AI показал LINQ-запрос вместо громоздкого цикла – разберите его, это обогатит ваш арсенал.
* Комбинируйте подходы. Иногда лучше спросить AI, иногда – просто подумать самому. Баланс приходит с опытом. Помните, что вы – разработчик, AI – инструмент. В итоге, важны ваши приобретённые знания.
* Пример рабочего процесса с Codex:
* Вы пишете код SimulationLoop. Не уверены, правильно ли сделали паузу. Вы спрашиваете: “Как правильно реализовать паузу в Unity с фиксированным шагом симуляции?”. AI отвечает, упоминая Time.timeScale и свой loop[[6]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=,%D0%BB%D1%83%D1%87%D1%88%D0%B5%20%D0%BA%D0%BE%D0%BD%D1%82%D1%80%D0%BE%D0%BB%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D1%82%D1%8C%20%D1%81%D0%B8%D0%BC%D1%83%D0%BB%D1%8F%D1%86%D0%B8%D1%8E%20%D1%8F%D0%B2%D0%BD%D1%8B%D0%BC%20%D1%86%D0%B8%D0%BA%D0%BB%D0%BE%D0%BC). Вы сравниваете с вашим методом. Решаете, что ваш метод ок, но добавляете совет AI про Animator.updateMode = UnscaledTime для UI анимаций.
* Вы запускаете, но видите: при паузе один тик всё же проскакивает. Путём логов не можете найти причину. Вы копируете код Update и спрашиваете AI: “Почему при таком коде SimulationLoop может выполнить лишний тик при установке паузы?”. AI анализирует и подсказывает, что, возможно, acc не обнуляется правильно или pausing не мгновенно. Вы находите баг – например, нужно сбросить acc при паузе. Исправляете.
* Позже, вам нужно быстро добавить юнит-тесты для Economy. Вы пишете вручную пару, затем думаете – а все ли случаи покрыли? Пишете Codex: “Сгенерируй дополнительные unit-тесты на класс Economy (TryReserve/Commit)”. Он предлагает тест на отрицательные суммы, на последовательные транзакции. Вы некоторые берёте, некоторые адаптируете.
* Вы чувствовали, что не до конца понимаете GOAP. В свободное время спрашиваете: “Объясни принцип GOAP на простом примере”. AI рассказывает (или даже на C# пример). Вы лучше разбираетесь в теме, углубляете свой проект.
* В конце проекта вы хотите навести порядок. Выделяете код монолитного метода AI, применяете /Refactor. AI разбивает на несколько функций – вы проверяете, все ли работает. Стало чище, поддерживаемее.
* Заключение: VS Code + Codex GPT – мощная связка. VS Code предоставляет лёгкую и гибкую IDE с отличными средствам навигации и отладки, а Codex – ускоряет получение информации и написание кода. Вместе они позволяют решать проблемы быстрее и учиться эффективнее. Но всегда держите под контролем процесс: проверяйте выводы AI, экспериментируйте самостоятельно. Со временем вы привыкнете формулировать запросы так, что AI будет давать именно то, что нужно.

Следуя этим рекомендациям и плану модулей, через квартал у вас будет прочная база знаний и навыков. Вы последовательно, шаг за шагом, прошли от основ к сложным системам, научились отлаживать и оптимизировать. Не бойтесь продолжать экспериментировать, задавать вопросы (и AI, и сообществу разработчиков) и совершенствовать проект. Успехов в разработке вашего RTS-проекта!

[[1]](file://file_000000003db061f4ab0ba4400c468bb3" \l ":~:text=%D0%A6%D0%B5%D0%BB%D1%8C%3A%20%D1%81%D0%B8%D0%BC%D1%83%D0%BB%D1%8F%D1%86%D0%B8%D1%8F%20%D0%B8%D0%B4%D1%91%D1%82%20%D1%84%D0%B8%D0%BA%D1%81%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D1%8B%D0%BC%20%D1%88%D0%B0%D0%B3%D0%BE%D0%BC,%D0%BE%D1%81%D1%82%D0%B0%D0%BD%D0%B0%D0%B2%D0%BB%D0%B8%D0%B2%D0%B0%D0%B5%D1%82%20%D1%81%D0%B8%D0%BC%D1%83%D0%BB%D1%8F%D1%86%D0%B8%D1%8E%2C%20%D0%BD%D0%BE%20%D0%BD%D0%B5%20UI) [[2]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=,%E2%80%94%20DOTS%2FEntities%20%D0%B4%D0%BB%D1%8F%20%D1%84%D0%B8%D0%BA%D1%81%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B3%D0%BE%20%D1%82%D0%B8%D0%BA%D0%B0) [[3]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=%D0%94%D0%B5%D0%BD%D1%8C%201%E2%80%932%20,%D1%82%D0%B8%D0%BA%D0%BE%D0%BC%20%D0%B8%20%D0%BF%D0%B0%D1%83%D0%B7%D0%BE%D0%B9) [[4]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=%60%60%60csharp%20public%20interface%20IClock%20,IsPaused%20%3F%200f%20%3A%20tick) [[5]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=public%20sealed%20class%20SimulationLoop%20%3A,IsPaused%20%3F%200f%20%3A%20tick) [[6]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=,%D0%BB%D1%83%D1%87%D1%88%D0%B5%20%D0%BA%D0%BE%D0%BD%D1%82%D1%80%D0%BE%D0%BB%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D1%82%D1%8C%20%D1%81%D0%B8%D0%BC%D1%83%D0%BB%D1%8F%D1%86%D0%B8%D1%8E%20%D1%8F%D0%B2%D0%BD%D1%8B%D0%BC%20%D1%86%D0%B8%D0%BA%D0%BB%D0%BE%D0%BC) [[7]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=while%20%28acc%20,Flush%28%29%3B) [[8]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=void%20Update%28%29%20,Tick%28tick) [[9]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=TacticalAI.Tick%28tick%29%3B%20EventBus.Flush%28%29%3B%20,IsPaused%20%3D%20v) [[18]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=3%29%20%D0%94%D0%B0%D0%BD%D0%BD%D1%8B%D0%B5%20%D0%B8%20%D0%BA%D0%BE%D0%BD%D1%84%D0%B8%D0%B3%D0%B8%20,%D1%81%D1%82%D0%B0%D1%80%D1%82%D0%B5%20%D0%BF%D1%80%D0%B5%D0%B2%D1%80%D0%B0%D1%89%D0%B0%D0%B5%D0%BC%20%D0%B2%20%D1%87%D0%B8%D1%81%D1%82%D1%8B%D0%B5%20%D0%BE%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D1%8B) [[19]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=%2F%2F%20Domain%2FTech%2FTech,string%3E%20Prereq%29%3B) [[20]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=%D0%9F%D0%BE%D1%87%D0%B5%D0%BC%D1%83%20%D1%82%D0%B0%D0%BA%3A%20SO%20%E2%80%94%20%D1%83%D0%B4%D0%BE%D0%B1%D0%B5%D0%BD,%D0%BA%D0%BE%D0%BF%D0%B8%D0%B8%20%E2%80%94%20%D0%B1%D0%B5%D0%B7%D0%BE%D0%BF%D0%B0%D1%81%D0%BD%D1%8B%20%D0%B8%20%D1%81%D0%B5%D1%80%D0%B8%D0%B0%D0%BB%D0%B8%D0%B7%D1%83%D0%B5%D0%BC%D1%8B) [[21]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=%2F%2F%20Data%2FTechs%2FTechDef.asset%20%28SO%29%20%5BCreateAssetMenu%28menuName%3D,Prereq%3B) [[29]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=,%D1%83%D0%BA%D1%80%D0%B5%D0%BF%D0%BB%D0%B5%D0%BD%D0%B8%D0%B9%20%D0%B8%20%C2%AB%D1%82%D0%BE%D1%87%D0%B5%D1%87%D0%BD%D1%83%D1%8E%20%D0%BF%D0%BE%D0%B4%D0%B4%D0%B5%D1%80%D0%B6%D0%BA%D1%83%C2%BB%20%D0%98%D0%98) [[30]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=%60%60%60csharp%20public%20interface%20IEconomy%20,float%20dt%29%3B) [[31]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=%2F%2F%20%D0%B2%D1%8B%D0%B7%D1%8B%D0%B2%D0%B0%D0%B5%D0%BC%20%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%BD%D1%8B%D0%B5%20%D1%82%D0%B8%D0%BA%D0%B8%20%D0%BF%D0%BE,Flush%28%29%3B) [[34]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=5%29%20%D0%98%D0%BD%D1%84%D0%B5%D0%BA%D1%86%D0%B8%D1%8F%20%28%D0%BC%D0%BE%D0%B4%D0%B5%D0%BB%D1%8C%20%D0%B7%D0%B0%D1%80%D0%B0%D0%B6%D0%B5%D0%BD%D0%B8%D1%8F%29%20,%D0%B4%D0%BE%D0%BB%D1%8F%20%D0%BD%D0%B0%D1%81%D0%B5%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F%2F%D0%BC%D0%BE%D0%B1%D0%B8%D0%BB%D1%8C%D0%BD%D0%BE%D1%81%D1%82%D1%8C) [[35]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=,level%3B%20public%20InfectionState%20state%3B) [[40]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=%D0%9C%D0%B8%D0%BD%D0%B8,0%29) [[41]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=%D0%9A%D0%BE%D0%BC%D0%B0%D0%BD%D0%B4%D0%B0%20%D1%81%D1%82%D1%80%D0%BE%D0%B8%D1%82%D0%B5%D0%BB%D1%8C%D1%81%D1%82%D0%B2%D0%B0%3A%20,Publish%28order%29%3B%20return%20order) [[42]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=public%20static%20class%20EventBus%20,Dequeue) [[43]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=public%20interface%20IBuildService%20,) [[44]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=public%20BuildOrder%20Place,Publish%28order%29%3B%20return%20order%3B) [[46]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=EventBus%20,%D1%81%D1%82%D1%80%D0%BE%D0%B8%D1%82%D0%B5%D0%BB%D1%8C%D1%81%D1%82%D0%B2%D0%BE%2C%20%D0%BD%D0%B0%D0%B7%D0%BD%D0%B0%D1%87%D0%B5%D0%BD%D0%B8%D0%B5%20%D0%B1%D1%8E%D0%B4%D0%B6%D0%B5%D1%82%D0%B0) [[47]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=public%20static%20void%20Subscribe,) [[49]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=%D0%A0%D0%B5%D0%BA%D0%BE%D0%BC%D0%B5%D0%BD%D0%B4%D0%BE%D0%B2%D0%B0%D0%BD%20%D0%B3%D0%B8%D0%B1%D1%80%D0%B8%D0%B4%20%D1%82%D1%80%D1%91%D1%85%20%D0%BF%D0%BE%D0%B4%D1%85%D0%BE%D0%B4%D0%BE%D0%B2%3A) [[50]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=1,%D0%94%D0%B5%D1%80%D0%B5%D0%B2%D1%8C%D1%8F%20%D1%85%D0%BE%D1%80%D0%BE%D1%88%D0%BE%20%D0%BE%D1%82%D0%BB%D0%B0%D0%B6%D0%B8%D0%B2%D0%B0%D1%8E%D1%82%D1%81%D1%8F%20%D0%B8%20%D1%87%D0%B8%D1%82%D0%B0%D0%B1%D0%B5%D0%BB%D1%8C%D0%BD%D1%8B) [[51]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=GOAP%20%28%D1%8D%D1%81%D0%BA%D0%B8%D0%B7%29%3A%20,Goal%20g%2C%20IWorldState%20w%29%3B) [[52]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=BT%20%28%D0%BF%D1%81%D0%B5%D0%B2%D0%B4%D0%BE%D0%BA%D0%BE%D0%B4%20%D1%83%D0%B7%D0%BB%D0%BE%D0%B2%29%3A%20,Patrol) [[53]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=%D0%A1%D0%BB%D0%BE%D0%B9%20%D0%B2%D0%BE%D1%81%D0%BF%D1%80%D0%B8%D1%8F%D1%82%D0%B8%D1%8F%3A%20,) [[59]](file://file_000000003db061f4ab0ba4400c468bb3#:~:text=Utility,5f%3B) document.txt

<file://file_000000003db061f4ab0ba4400c468bb3>

[[10]](file://file_00000000e2c461f4bb2cbf5057f4f2ac#:~:text=2,%D1%86%D0%B8%D0%BA%D0%BB%D1%83) [[17]](file://file_00000000e2c461f4bb2cbf5057f4f2ac#:~:text=%EE%88%80cite%EE%88%82turn6search6%EE%88%82turn6search0%EE%88%81.%20,%D0%B2%D1%8B%D0%B8%D0%B3%D1%80%D1%8B%D1%88%20%D0%BD%D0%B0%20%D0%B2%D0%B0%D1%88%D0%B5%D0%B9%20%D1%81%D1%86%D0%B5%D0%BD%D0%B5%20%EE%88%80cite%EE%88%82turn3search13%EE%88%82turn3search0%EE%88%82turn3search1%EE%88%81) [[22]](file://file_00000000e2c461f4bb2cbf5057f4f2ac#:~:text=%2A%2A%D0%A7%D1%82%D0%BE%20%D0%B4%D0%BE%D0%B1%D0%B0%D0%B2%D0%B8%D1%82%D1%8C%20%28%D0%BC%D0%B5%D1%85%D0%B0%D0%BD%D0%B8%D0%BA%D0%B8%29%3A%2A%2A%20,%D0%BC%D0%B8%D0%B3%D1%80%D0%B0%D1%86%D0%B8%D0%B8%20%D0%BF%D1%80%D0%B8%20%D0%B7%D0%B0%D0%B3%D1%80%D1%83%D0%B7%D0%BA%D0%B5) [[27]](file://file_00000000e2c461f4bb2cbf5057f4f2ac#:~:text=,%D0%BF%D0%BE%20%D1%81%D0%BF%D0%BE%D1%81%D0%BE%D0%B1%D0%B0%D0%BC%20%D0%B7%D0%B0%D0%B3%D1%80%D1%83%D0%B7%D0%BA%D0%B8%2F%D1%87%D0%B0%D1%81%D1%82%D0%BE%D1%82%D0%B5%20%D0%B8%D1%81%D0%BF%D0%BE%D0%BB%D1%8C%D0%B7%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F%20%EE%88%80cite%EE%88%82turn2search10%EE%88%82turn2search16%EE%88%81) [[45]](file://file_00000000e2c461f4bb2cbf5057f4f2ac#:~:text=,50%20%D0%BA%D0%BB%D0%B5%D1%82%D0%BA%D0%B0%D1%85%C2%BB%20%E2%86%92%20%D0%BE%D0%B4%D0%B8%D0%BD%20%D0%B1%D0%B0%D1%82%D1%87) [[48]](file://file_00000000e2c461f4bb2cbf5057f4f2ac#:~:text=%2A%2A%D0%9A%D0%B0%D0%BA%20%D1%83%D1%81%D0%BA%D0%BE%D1%80%D0%B8%D1%82%D1%8C%3A%2A%2A%20,%D0%BF%D0%B0%D0%BA%D0%B5%D1%82%D0%BD%D0%B0%D1%8F%20%D0%BE%D0%B1%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%BA%D0%B0%2F%D1%84%D0%BB%D0%B0%D1%88%20%D0%BF%D0%BE%20%D0%BA%D0%BE%D0%BD%D1%86%D1%83%20%D1%82%D0%B8%D0%BA%D0%B0) [[57]](file://file_00000000e2c461f4bb2cbf5057f4f2ac#:~:text=%2A%2A%D0%9A%D0%B0%D0%BA%20%D1%83%D1%81%D0%BA%D0%BE%D1%80%D0%B8%D1%82%D1%8C%3A%2A%2A%20,%D0%BE%D0%B1%D0%BD%D0%BE%D0%B2%D0%BB%D1%8F%D1%82%D1%8C%20%D1%82%D0%BE%D0%BB%D1%8C%D0%BA%D0%BE%20%D0%BF%D1%80%D0%B8%20%D0%BF%D0%B5%D1%80%D0%B5%D1%81%D1%82%D1%80%D0%BE%D0%B9%D0%BA%D0%B5%20%D0%BA%D0%B0%D1%80%D1%82%D1%8B) [[58]](file://file_00000000e2c461f4bb2cbf5057f4f2ac#:~:text=,%D0%B8%20%D0%B1%D0%BE%D0%B3%D0%B0%D1%82%D0%B0%20%D0%BD%D0%B0%20%D1%8D%D0%B2%D1%80%D0%B8%D1%81%D1%82%D0%B8%D0%BA%D0%B8%20%EE%88%80cite%EE%88%82turn1search1%EE%88%81) [[60]](file://file_00000000e2c461f4bb2cbf5057f4f2ac#:~:text=,%D0%B8%20%D0%B1%D0%BE%D0%B3%D0%B0%D1%82%D0%B0%20%D0%BD%D0%B0%20%D1%8D%D0%B2%D1%80%D0%B8%D1%81%D1%82%D0%B8%D0%BA%D0%B8%20%EE%88%80cite%EE%88%82turn1search1%EE%88%81) [[61]](file://file_00000000e2c461f4bb2cbf5057f4f2ac#:~:text=,%D0%BF%D1%80%D0%BE%D1%85%D0%BE%D0%B4%20%D1%87%D0%B5%D1%80%D0%B5%D0%B7%20%D0%B7%D0%B0%D1%80%D0%B0%D0%B6%D1%91%D0%BD%D0%BD%D1%8B%D0%B5%20%C2%AB%D0%BA%D1%80%D0%B0%D1%81%D0%BD%D1%8B%D0%B5%C2%BB%20%D0%B7%D0%BE%D0%BD%D1%8B) [[64]](file://file_00000000e2c461f4bb2cbf5057f4f2ac#:~:text=%2A%2A%D0%9A%D0%B0%D0%BA%20%D1%83%D1%81%D0%BA%D0%BE%D1%80%D0%B8%D1%82%D1%8C%3A%2A%2A%20,%D0%BF%D0%B0%D1%80%20%C2%AB%D1%80%D0%B5%D0%B3%D0%B8%D0%BE%D0%BD%E2%86%92%D1%80%D0%B5%D0%B3%D0%B8%D0%BE%D0%BD%C2%BB%20%D0%B8%20%D0%BA%D1%80%D0%B0%D1%82%D1%87%D0%B0%D0%B9%D1%88%D0%B8%D1%85%20%C2%AB%D0%BA%D0%BE%D1%80%D0%B8%D0%B4%D0%BE%D1%80%D0%BE%D0%B2%C2%BB) [[66]](file://file_00000000e2c461f4bb2cbf5057f4f2ac#:~:text=%D0%9A%D0%BE%D1%80%D0%BE%D1%82%D0%BA%D0%B8%D0%B5%20%D0%B4%D0%BE%D0%BF,%D0%B8%D0%BD%D0%B0%D1%87%D0%B5%20%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D1%8C%20%D1%80%D0%B0%D1%81%D0%BF%D1%83%D1%85%D0%B0%D0%B5%D1%82) [[67]](file://file_00000000e2c461f4bb2cbf5057f4f2ac#:~:text=%E2%80%94%20Utility,T%3E%60%29%20%EE%88%80cite%EE%88%82turn6search6%EE%88%81) document-2.txt

<file://file_00000000e2c461f4bb2cbf5057f4f2ac>

[[11]](file://file_0000000069fc624685d94f067e42a757#:~:text=3%29%20%D0%9F%D0%B0%D1%83%D0%B7%D0%B0%3A%20%D0%BF%D1%80%D0%BE%D0%B2%D0%B5%D1%80%D1%8C%2C%20%D1%87%D1%82%D0%BE%20,%EE%88%80cite%EE%88%82turn0search12%EE%88%81) [[12]](file://file_0000000069fc624685d94f067e42a757#:~:text=,tick) [[13]](file://file_0000000069fc624685d94f067e42a757#:~:text=6%29%20%D0%9F%D1%80%D0%BE%D1%84%D0%B8%D0%BB%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5%3A%20,%EE%88%80cite%EE%88%82turn0search12%EE%88%81) [[14]](file://file_0000000069fc624685d94f067e42a757#:~:text=,%D0%A3%D0%B1%D0%B5%D0%B4%D0%B8%D1%81%D1%8C%2C%20%D1%87%D1%82%D0%BE%20%D0%B8%D0%B3%D1%80%D0%BE%D0%B2%D0%BE%D0%B9%20%D0%BA%D0%BE%D0%B4) [[15]](file://file_0000000069fc624685d94f067e42a757#:~:text=%EE%88%80cite%EE%88%82turn0search12%EE%88%81) [[16]](file://file_0000000069fc624685d94f067e42a757#:~:text=,%D0%B2%20%D0%BA%D0%BE%D0%BD%D1%86%D0%B5%20%D1%82%D0%B8%D0%BA%D0%B0) [[23]](file://file_0000000069fc624685d94f067e42a757#:~:text=1%29%20,%D0%BA%D0%BE%D1%80%D1%80%D0%B5%D0%BA%D1%82%D0%BD%D0%BE%20%D0%B2%D0%BE%D1%81%D1%81%D1%82%D0%B0%D0%BD%D0%B0%D0%B2%D0%BB%D0%B8%D0%B2%D0%B0%D0%B5%D1%82%20%D0%BF%D0%BE%D0%B4%D1%82%D0%B8%D0%BF%D1%8B) [[24]](file://file_0000000069fc624685d94f067e42a757#:~:text=%D0%B0%D0%B4%D0%B0%D0%BF%D1%82%D0%B5%D1%80%2C%20%D0%BB%D0%B8%D0%B1%D0%BE%20%D1%81%D1%82%D0%BE%D1%80%D0%BE%D0%BD%D0%BD%D0%B8%D0%B9%20JSON,%D0%B7%D0%B0%D0%B3%D1%80%D1%83%D0%B7%D0%BA%D0%B5%20%E2%80%94%20%D0%BC%D0%B8%D0%B3%D1%80%D0%B0%D1%86%D0%B8%D1%8F%20%D0%BF%D0%BE%20%D1%81%D0%B2%D0%B8%D1%82%D1%87%D1%83) [[25]](file://file_0000000069fc624685d94f067e42a757#:~:text=%21AssetDatabase.Contains%28obj%29%29,%D0%BA%D0%BE%D1%80%D1%80%D0%B5%D0%BA%D1%82%D0%BD%D0%BE%20%D0%B2%D0%BE%D1%81%D1%81%D1%82%D0%B0%D0%BD%D0%B0%D0%B2%D0%BB%D0%B8%D0%B2%D0%B0%D0%B5%D1%82%20%D0%BF%D0%BE%D0%B4%D1%82%D0%B8%D0%BF%D1%8B.%20%EE%88%80cite%EE%88%82turn0search15%EE%88%81) [[26]](file://file_0000000069fc624685d94f067e42a757#:~:text=%60,%EE%88%80cite%EE%88%82turn1search6%EE%88%81) [[28]](file://file_0000000069fc624685d94f067e42a757#:~:text=,%EE%88%80cite%EE%88%82turn1search10%EE%88%81) [[32]](file://file_0000000069fc624685d94f067e42a757#:~:text=1%29%20Unit,%C2%AB%D0%B4%D0%B5%D0%BD%D1%8C%D0%B3%D0%B8%20%D0%BD%D0%B5%20%D0%B8%D1%81%D1%87%D0%B5%D0%B7%D0%B0%D1%8E%D1%82%2F%D0%BD%D0%B5%20%D0%BF%D0%BE%D1%8F%D0%B2%D0%BB%D1%8F%D1%8E%D1%82%D1%81%D1%8F%C2%BB) [[33]](file://file_0000000069fc624685d94f067e42a757#:~:text=%E2%86%92%20%D0%B8%D1%82%D0%BE%D0%B3%D0%BE%D0%B2%D1%8B%D0%B5%20%D0%B1%D0%B0%D0%BB%D0%B0%D0%BD%D1%81%D1%8B%20%D0%BF%D1%80%D0%B5%D0%B4%D1%81%D0%BA%D0%B0%D0%B7%D1%83%D0%B5%D0%BC%D1%8B,%D0%BF%D1%80%D0%BE%D0%B2%D0%B5%D1%80%D1%8C%2C%20%D1%87%D1%82%D0%BE%20%D0%BD%D0%B5%D1%82%20%D0%B4%D1%80%D0%BE%D0%B6%D0%B0%D0%BD%D0%B8%D1%8F%20%D0%BF%D1%80%D0%B8%D0%BE%D1%80%D0%B8%D1%82%D0%B5%D1%82%D0%BE%D0%B2) [[36]](file://file_0000000069fc624685d94f067e42a757#:~:text=1,%D0%BA%D0%B0%D1%80%D1%82%D1%8B%3A%20%D0%BF%D0%BE%D0%BC%D0%BE%D0%B3%D0%B0%D0%B5%D1%82%20%C2%AB%D0%B3%D0%BB%D0%B0%D0%B7%D0%BE%D0%BC%C2%BB%20%D0%BB%D0%BE%D0%B2%D0%B8%D1%82%D1%8C%20%D1%81%D0%BA%D0%B0%D1%87%D0%BA%D0%B8%2F%D0%B7%D0%B0%D1%81%D1%82%D0%BE%D0%B8) [[37]](file://file_0000000069fc624685d94f067e42a757#:~:text=%EE%88%80cite%EE%88%82turn4search0%EE%88%81%202,%D0%BA%D0%B0%D1%80%D1%82%D1%8B%3A%20%D0%BF%D0%BE%D0%BC%D0%BE%D0%B3%D0%B0%D0%B5%D1%82%20%C2%AB%D0%B3%D0%BB%D0%B0%D0%B7%D0%BE%D0%BC%C2%BB%20%D0%BB%D0%BE%D0%B2%D0%B8%D1%82%D1%8C%20%D1%81%D0%BA%D0%B0%D1%87%D0%BA%D0%B8%2F%D0%B7%D0%B0%D1%81%D1%82%D0%BE%D0%B8) [[38]](file://file_0000000069fc624685d94f067e42a757#:~:text=%E2%80%94%20%D1%82%D0%B5%D1%81%D1%82%20%C2%AB%D0%B7%D0%BD%D0%B0%D1%87%D0%B5%D0%BD%D0%B8%D0%B5%20%D0%BD%D0%B5%20%D0%B7%D0%B0%D1%82%D0%B8%D1%80%D0%B0%D0%B5%D1%82%D1%81%D1%8F,%D0%BA%D0%B0%D1%80%D1%82%D1%8B%3A%20%D0%BF%D0%BE%D0%BC%D0%BE%D0%B3%D0%B0%D0%B5%D1%82%20%C2%AB%D0%B3%D0%BB%D0%B0%D0%B7%D0%BE%D0%BC%C2%BB%20%D0%BB%D0%BE%D0%B2%D0%B8%D1%82%D1%8C%20%D1%81%D0%BA%D0%B0%D1%87%D0%BA%D0%B8%2F%D0%B7%D0%B0%D1%81%D1%82%D0%BE%D0%B8) [[39]](file://file_0000000069fc624685d94f067e42a757#:~:text=,%D1%80%D0%B0%D0%B7%D0%BC%D0%B5%D1%80%20%D1%84%D1%80%D0%BE%D0%BD%D1%82%D0%B0%20%D0%B8%20%D0%B2%D1%80%D0%B5%D0%BC%D1%8F%20%D1%88%D0%B0%D0%B3%D0%B0) [[54]](file://file_0000000069fc624685d94f067e42a757#:~:text=,%D1%80%D0%B5%D0%B6%D0%B8%D0%BC%D0%B5) [[55]](file://file_0000000069fc624685d94f067e42a757#:~:text=,%EE%88%80cite%EE%88%82turn2search2%EE%88%81) [[56]](file://file_0000000069fc624685d94f067e42a757#:~:text=,%EE%88%80cite%EE%88%82turn2search5%EE%88%81) [[62]](file://file_0000000069fc624685d94f067e42a757#:~:text=,%EE%88%80cite%EE%88%82turn3search5%EE%88%82turn3search20%EE%88%81) [[63]](file://file_0000000069fc624685d94f067e42a757#:~:text=,%EE%88%80cite%EE%88%82turn3search0%EE%88%82turn3search3%EE%88%81) [[65]](file://file_0000000069fc624685d94f067e42a757#:~:text=3%29%20DOTS%2FEntities%20%E2%80%94%20%D0%A4%D0%B8%D0%BA%D1%81,%EE%88%80cite%EE%88%82turn0search2%EE%88%82turn1search19%EE%88%81) document-3.txt

<file://file_0000000069fc624685d94f067e42a757>

[[68]](https://marketplace.visualstudio.com/items?itemName=DanielSanMedium.dscodegpt#:~:text=Chat%20Commands) [[69]](https://marketplace.visualstudio.com/items?itemName=DanielSanMedium.dscodegpt#:~:text=,context%20for%20the%20AI%20agent) CodeGPT: Chat & AI Agents - Visual Studio Marketplace

<https://marketplace.visualstudio.com/items?itemName=DanielSanMedium.dscodegpt>