Генерация поля боя

в Heroes of Might and Magic III

Технический разбор алгоритма:
от клетки карты приключений до пикселей на экране

С анализом исходников **VCMI**

Открытая реализация движка Heroes III

Апрель 2026

Содержание

Этот гайд собран из диалога по мотивам вопросов о внутреннем устройстве боевой системы «Героев 3». Где возможно, приведены ссылки на конкретные файлы и строки в открытом репозитории VCMI — fan-made-реализации движка, которая воспроизводит поведение оригинала.

1. Общий алгоритм генерации поля боя

Поле боя в «Героях 3» не случайное в полном смысле слова — оно **детерминированно** вычисляется из параметров той клетки карты приключений, на которой стоит обороняющийся. Дальше разберём, из чего оно состоит.

Что определяет ландшафт поля боя

Тип почвы (трава, грязь, песок, болото, снег, каменистая, подземка, лава, вода) берётся с клетки, где стоит защитник. Поэтому если твой герой и герой противника стоят вплотную, но на стыке разных ландшафтов, игра сгенерирует поле боя с ландшафтом обороняющейся стороны. Тот же принцип работает при атаке нейтралов или захвате охраняемого объекта — берётся клетка входа.

Поверх базового ландшафта могут «лечь» особые поля — Магические равнины, Проклятая земля, Клеверное поле, Злой туман, Святая земля и так далее. Они перекрывают обычный ландшафт и меняют правила боя.

Как размещаются препятствия — ключевой момент

Алгоритм использует три числа: X, Y и Z (горизонталь, вертикаль и уровень карты — поверхность или подземелье) плюс тип земли. На любой карте на одинаковых координатах при одинаковом типе земли будет одно и то же поле боя с идентичными препятствиями. То есть это не «бросок костей» в момент начала боя, а чистая функция от координат: если дважды атаковать нейтралов на клетке (45, 72, поверхность, трава) — получишь два раза одно и то же расположение пней, камней и озёр.

Это важное свойство для онлайн-игроков: существуют программы, которые по координатам и типу земли показывают расстановку препятствий заранее — как раз потому, что алгоритм повторяемый.

Два типа препятствий

Статичные препятствия — большие озёра, трещины и подобные — нельзя убрать заклинанием «Устранение преград». **Удаляемые** (брёвна, камни, пни) — можно. Набор возможных препятствий привязан к типу земли: на снегу не появятся пальмы, в лаве — трава, и так далее.

Сколько всего уникальных полей

Фанатами подсчитано, что в игре существует **393 984** уникальных боевых поля, образованных сочетанием типа земли и случайных препятствий. Не все они равноценны по балансу: на некоторых полях есть клетки, абсолютно недоступные для атаки в ближнем бою — но туда можно поместить летающее

существо или любое другое при помощи телепорта. Есть также «закутки», в которые не могут попасть двухгексовые существа, но одногексовые заходят без проблем.

Именно поэтому у серьёзных игроков генерация поля иногда становится частью тактики — выманить противника так, чтобы бой произошёл на удобной координате.

Сетка

Само поле боя — шестиугольная сетка **15 клеток в ширину × 11 в высоту** (плюс два столбца у краёв для начальной расстановки отрядов, итого 17 столбцов). Препятствия занимают от одного до нескольких гексов, и их границы с точностью до гекса определяют, кто куда пройдёт и куда долетит стрела без штрафа.

2. Как генерируются препятствия

Алгоритм полностью раскопан фанатами и воспроизведён в VCMI (открытом движке H3), так что можно говорить конкретно.

Общая схема

На входе у алгоритма четыре параметра: тип земли и координаты X, Y, Z клетки обороняющегося. Из этих четырёх чисел детерминированно вычисляется «семя» (seed) для генератора псевдослучайных чисел. Дальше уже этим ГПСЧ «бросают кости» — но поскольку семя всегда одно и то же для одной и той же клетки, результат тоже всегда один и тот же. Это чистая функция без реальной случайности.

Как устроена сетка

Поле боя — прямоугольник из **17 столбцов × 11 рядов** шестиугольников, итого 187 гексов с индексами 0…186. Крайний левый столбец (0, 17, 34, …) и крайний правый (16, 33, 50, …) — это зоны стартовой расстановки отрядов, в них препятствий не бывает. Получается рабочая область 15 × 11 = 165 клеток, куда алгоритм может что-то поставить.

Внутри формата препятствий из VCMI используется такая арифметика: вычитание 17 из номера гекса перемещает на одну клетку выше — то есть координаты препятствия задаются относительно левого нижнего угла, а ряды отсчитываются шагом в 17. Так описываются многоклеточные препятствия.

Два типа препятствий в генераторе

Препятствия делятся по двум независимым осям.

По способу размещения на поле

- **Обычные (non-absolute)** — хранят список блокируемых гексов относительно своего «якорного» угла. Могут быть размножены и помещены в разные точки поля. Именно они дают основное разнообразие полей боя.
- **Абсолютные (absolute)** — привязаны к конкретным координатам на поле. На одном поле может появиться только одно такое препятствие. Это обычно крупные «ландшафтные» вещи — большое озеро, огромная трещина.

По взаимодействию с магией

- **Статичные** — нельзя убрать заклинанием «Устранение преград».
- **Удаляемые** — можно.

Что привязано к каждому препятствию

У каждого препятствия в данных игры прописаны:

- список разрешённых типов почв (*allowedTerrains*);
- список специальных полей боя (*specialBattlefields*);
- флаг *absolute*;
- ширина и высота в гексах;
- список блокируемых клеток (*blockedTiles*);
- имя графического файла и флаг «рисовать поверх юнитов».

Когда алгоритм начинает заполнять поле, он сперва отсекает весь «каталог» препятствий по типу земли — на снегу не может появиться пальма, в подземелье не будет травянистой кочки.

Сам процесс размещения

Происходит примерно так:

1. Из seed инициализируется ГПСЧ.

2. Выясняется тип поля — базовый ландшафт или «накладной» (Магические равнины, Проклятая земля, Клеверное поле, Злой туман, Святая земля). Для спецполей используется свой список препятствий.

3. С определённой вероятностью выбирается **одно** абсолютное препятствие из пула для этого поля и ставится в свою фиксированную позицию.

4. Выбирается количество обычных препятствий (число ограничено, чтобы поле не превращалось в лабиринт).

5. Для каждого обычного препятствия: случайно выбирается экземпляр из каталога, случайно выбирается позиция (якорный гекс). Проверяется, что все клетки из *blockedTiles* этого препятствия **(а)** лежат внутри поля, **(б)** не попадают в стартовые столбцы юнитов, **(в)** не пересекаются с уже поставленными препятствиями. Если проверка не прошла — позиция бракуется и берётся следующая. Если препятствие вообще некуда поставить, оно пропускается.

6. Собранный набор (индексы гексов + графика) отправляется в боевой движок.

Почему это заметно в игре

Из этой схемы вырастают несколько следствий, которые ощущают игроки:

- Поле заранее предсказуемо — для онлайн-игр существуют утилиты, которые по X, Y и ландшафту показывают расстановку препятствий до боя.
- На некоторых полях появляются клетки-«кармашки», недоступные ближнему бою вообще, или закутки, куда пролезает одногексовый юнит, но не двухгексовый — это следствие того, что генератор не проверяет глобальную связность поля, только локальные коллизии при размещении.
- Огромное число комбинаций (≈394 тысячи уникальных полей) даёт множество частных случаев, часть из которых ломает баланс.

3. Как VCMI определяет, что гекс занят препятствием

Здесь разберём конкретный код VCMI (ветка *develop*), чтобы увидеть не только идею, но и её реализацию «в железе».

Двухуровневая модель: «тип» и «экземпляр»

В VCMI (как и в оригинальных «Героях») препятствие — это пара объектов:

- **ObstacleInfo** (*lib/ObstacleHandler.h*) — «шаблон» препятствия: размеры, список заблокированных клеток, разрешённые ландшафты, анимация. Загружается один раз из *config/obstacles.json*.
- **CObstacleInstance** (*lib/battle/CObstacleInstance.h*) — конкретное размещение на поле боя: позиция *pos* (якорь — левый нижний угол) и *ID* шаблона.

Сам ObstacleInfo хранит главное:

```
si32 width;                       // сколько клеток нужно справа
si32 height;                      // сколько клеток нужно сверху
std::vector<si16> blockedTiles;   // смещения относительно якоря
bool isAbsoluteObstacle;
std::vector<TerrainId> allowedTerrains;
```

Система координат поля

В *lib/battle/BattleHex.h* объявлено: BFIELD_WIDTH = 17, BFIELD_HEIGHT = 11, всего 187 гексов. Каждому гексу дан линейный индекс по формуле:

```
hex = x + y * 17;       // упаковка
x   = hex % 17;         // распаковка
y   = hex / 17;
```

Крайние столбцы (x = 0 и x = 16) зарезервированы под стартовую расстановку отрядов — это видно в методе *isAvailable()*, который проверяет *getX() > 0 && getX() < 16*.

Ключевая функция: ObstacleInfo::getBlocked

Вот сердце алгоритма — *lib/ObstacleHandler.cpp*, строки 58–80 — именно она отвечает на вопрос «какие гексы входят в препятствие»:

```
BattleHexArray ObstacleInfo::getBlocked(const BattleHex & hex) const
{
    if(isAbsoluteObstacle)
    {
        assert(!hex.isValid());
        return BattleHexArray(blockedTiles);
    }

    BattleHexArray ret;
    for(int offset : blockedTiles)
    {
```

```
        BattleHex toBlock = hex.toInt() + offset;
        if((hex.getY() & 1) && !(toBlock.getY() & 1))
            toBlock += BattleHex::LEFT;

        if(!toBlock.isValid())
            logGlobal->error("Misplaced obstacle!");
        else
            ret.insert(toBlock);
    }
    return ret;
}
```

Абсолютные препятствия

Большие озёра, трещины. У них *blockedTiles* — это уже готовые абсолютные индексы гексов, никаких вычислений не нужно. Параметр *hex* при вызове для такого препятствия специально передают невалидным — ассерт это проверяет.

Обычные препятствия

У них *blockedTiles* — список *смещений* от якорного гекса. Пример из *obstacles.json*:

```
"2": {
    "allowedTerrains" : ["dirt"],
    "width"  : 4,
    "height" : 2,
    "blockedTiles" :  [0, 1, -14, -15, -16],
    ...
}
```

[0, 1] — сам якорь и клетка справа (тот же ряд). [−14, −15, −16] — три клетки в ряду выше (−17 = прямо сверху, поэтому −16 и −15 — сверху-справа, −14 — ещё правее).

Почему там поправка с getY() & 1

Это самое хитрое место. Сетка «Героев» — это **offset-hex**: чётные и нечётные ряды сдвинуты относительно друг друга на пол-гекса. Из-за этого простое арифметическое смещение вроде −17 ведёт себя по-разному в зависимости от чётности ряда:

- Якорь в **чётном** ряду, offset = −17 → попадаешь в сосед **сверху-слева**.
- Якорь в **нечётном** ряду, offset = −17 → попадаешь в сосед **сверху-справа**.

Это видно в *moveInDirection* (*BattleHex.h*):

```
case TOP_LEFT:
    setXY((y % 2) ? x - 1 : x, y - 1, ...);
case TOP_RIGHT:
    setXY((y % 2) ? x : x + 1, y - 1, ...);
```

Дизайнеры договорились, что смещения в конфиге пишутся **в системе отсчёта чётного ряда**. Поэтому если препятствие положили в нечётный ряд и смещение увело в чётный (то есть мы прыгнули вверх через границу чётности), нужно скомпенсировать: сдвинуть результат на одну клетку влево. Это и делает строка:

```
if((hex.getY() & 1) && !(toBlock.getY() & 1))
    toBlock += BattleHex::LEFT;
```

Комментарий в *config/obstacles.json* это проговаривает: смещение −16 при такой схеме **всегда** означает «верхний-правый гекс», независимо от того, в какой ряд поставили препятствие.

Обратный вопрос: «эта клетка занята препятствием?»

Для этого есть *CBattleInfoCallback::battleGetAllObstaclesOnPos* (*lib/battle/CBattleInfoCallback.cpp*, строки 1048–1061). Логика прямолинейная — пробежать по всем препятствиям на поле и спросить каждое:

```
for(auto & obs : battleGetAllObstacles())
{
    if(obs->getBlockedTiles().contains(tile)
            || (!onlyBlocking && obs->getAffectedTiles().contains(tile)))
    {
        obstacles.push_back(obs);
    }
}
```

getBlockedTiles() у экземпляра просто вызывает *getAffectedTiles()*, который для обычных и абсолютных препятствий сводится к *getInfo().getBlocked(pos)* — то есть к той самой функции, которую мы разобрали выше. Кэширования нет: при каждом запросе список пересчитывается заново. Но препятствий на поле обычно меньше десятка, а у каждого — 1–6 клеток, так что это не узкое место.

Итого

Чтобы проверить, входит ли гекс в препятствие, VCMI делает следующее:

1. Для каждого размещённого препятствия берёт его якорную позицию и массив смещений из шаблона.

2. Для каждого смещения складывает его с якорем (целочисленно, в линейных индексах).

3. Если якорь в нечётном ряду, а результат — в чётном, сдвигает результат на одну клетку влево (компенсация offset-hex-сетки).

4. Сравнивает полученный список с интересующей клеткой.

Это прямой и довольно красивый подход: вся геометрия препятствия закодирована одним массивом int-смещений, а поправка на чётность ряда — единственное место, где «вылезает» специфика шестиугольной сетки.

4. Фон поля боя и «безопасная зона»

«Безопасная зона» для препятствий и гексов работает не случайно — она жёстко захардкожена в нескольких местах одновременно. Разберём всю цепочку.

Фон — это просто готовая картинка, которую кладут в (0, 0)

В *client/battle/BattleFieldController.cpp*, строки 124–140, движок берёт тип поля (из клетки карты приключений) и загружает соответствующий BMP:

```
auto bfieldType = owner.getBattle()->battleGetBattlefieldType();
background = ENGINE->renderHandler().loadImage(
    bfieldType.getInfo()->graphics, EImageBlitMode::OPAQUE);

pos.w = background->width();
pos.h = background->height();
```

Таблица соответствия лежит в *config/battlefields.json*:

```
"sand_shore":   { "graphics" : "CMBKBCH.BMP"  },
"grass_hills":  { "graphics" : "CMBKGRMT.BMP" },
"lava":         { "graphics" : "CMBKLAVA.BMP" },
"magic_plains": { "graphics" : "CMBKMAG.BMP",
                  "isSpecial" : true, ... },
```

Это оригинальные файлы из H3 (CMBK*.BMP — **C**o**m**bat **B**ac**k**ground), все размером 800×556. Фон рисуется в левый верхний угол области боя (*showBackgroundImage* на строке 237 — *canvas.draw(background, Point(0, 0))*). Никакого позиционирования фона как такового нет: он и есть вся область.

Гексы попадают в пиксели по жёсткой формуле

Ключевая функция — *hexPositionLocal* (*BattleFieldController.cpp*, строка 620):

```
Rect BattleFieldController::hexPositionLocal(const BattleHex & hex) const
{
    int x = 14 + ((hex.getY())%2==0 ? 22 : 0) + 44*hex.getX();
    int y = 86 + 42 *hex.getY();
    ...
}
```

Расшифруем константы:

- **14** — левый отступ от края фона до первого столбца гексов.
- **86** — верхний отступ (художники рисовали там «небо» или верхушки деревьев).
- **44** — шаг по горизонтали между соседями в одном ряду.
- **42** — шаг по вертикали между рядами.

- **22** (= 44/2) — сдвиг чётных рядов (Y = 0, 2, 4, …) вправо для offset-hex-сетки.

Подставим крайние значения:

- Гекс (0, 0) → пиксель (14 + 22 + 0, 86) = **(36, 86)**.
- Гекс (16, 10) → пиксель (14 + 22 + 44·16, 86 + 42·10) = **(740, 506)**.

Сетка укладывается примерно в прямоугольник от (14, 86) до (784, 548) внутри фона 800×556. Обрати внимание: эти числа — часть **контракта между движком и художниками**. Когда в 1999-м рисовали CMBKGRMT.BMP и компанию, авторы знали: на пикселе (14, 86) начнётся гекс (0, 0), дальше шаг 44×42. Они под это и рисовали — деревья, скалы и прочая «декорация» либо попадают в свои ячейки, либо специально размещаются на полях за пределами игровой сетки. «Безопасная зона» для гексов — не результат вычислений, а результат ручной художественной работы под известную схему.

Препятствия — двойная защита от выхода за границы

Теперь самое важное — сама генерация в *lib/battle/BattleInfo.cpp*, строки 199–290. Там есть *validPosition* — лямбда, через которую проходит каждая кандидатная позиция:

```
auto validPosition = [&](const BattleHex & pos) -> bool
{
    if(obi.height >= pos.getY())        return false;  // (A)
    if(pos.getX() == 0)                 return false;  // (B)
    if(pos.getX() + obi.width > 15)     return false;  // (C)
    if(blockedTiles.contains(pos))      return false;  // (D)

    for(const BattleHex & blocked : obi.getBlocked(pos))
    {
        if(tileAccessibility[blocked.toInt()]
             == EAccessibility::UNAVAILABLE) return false;  // (E)
        if(blockedTiles.contains(blocked))   return false;  // (F)
        int x = blocked.getX();
        if(x <= 2 || x >= 14)                return false;  // (G) ← !
    }
    return true;
};

RangeGenerator posgenerator(18, 168, ourRand);              // (H)
```

Каждая проверка — отдельная гарантия:

(A) Якорь не должен быть слишком близко к верхнему краю: для препятствия высотой *height* рядов нужны *height* строк над ним. Иначе *getBlocked* выкатится за верхнюю границу с отрицательным индексом.

(B) Якорь не в нулевом столбце (левый край).

(C) Правая граница препятствия (X + width) не должна вылезти за столбец 15. Таким образом 16-й столбец (защитник) заведомо остаётся чистым.

(D) Якорная клетка не занята другим препятствием.

(E) На особых полях (вроде морского боя «корабль против корабля») часть гексов помечена как UNAVAILABLE — туда вообще нельзя.

(F) Ни одна из блокируемых клеток не пересекается с уже поставленным препятствием.

(G) **Главное:** каждая заблокированная клетка должна лежать в столбцах **3…13** включительно. Столбцы 0–2 и 14–16 — обязательная «буферная зона» вокруг стартовых позиций отрядов.

(H) Сам генератор позиций выбирает индекс гекса не из всего диапазона 0…186, а только из **18…168**. 18 = начало второго ряда (17 — это Y = 1, X = 0; +1 = первая валидная клетка). 168 — примерно предпоследний ряд. То есть верхний и нижний ряды тоже заранее отрезаны.

Комбинация **(G)+(H)** означает: любое обычное препятствие поместится строго внутри прямоугольника из столбцов 3–13 и в средних рядах. На поле 17×11 это внутренняя область примерно 11×9 клеток — в пикселях от ~(146, 128) до ~(620, 464). Всё остальное — отступы для юнитов и UI.

Отрисовка препятствия привязана к тому же гексу

В *client/battle/BattleObstacleController.cpp*, строка 197:

```
Point BattleObstacleController::getObstaclePosition(
    std::shared_ptr<IImage> image, const CObstacleInstance & obstacle)
{
    int offset = obstacle.getAnimationYOffset(image->height());
    Rect r = owner.fieldController->hexPositionLocal(obstacle.pos);
    r.y += 42 - image->height() + offset;
    return r.topLeft();
}
```

Спрайт препятствия берёт пиксельные координаты своего якорного гекса (через ту же *hexPositionLocal*) и выравнивается **по низу этой клетки**: *r.y += 42 − image->height()*. То есть высокая картинка (дерево, камень) «вырастает вверх» из клетки, где стоит её якорь. Плюс в *getAnimationYOffset* (*lib/battle/CObstacleInstance.cpp*, строка 75) есть подгонка ±42 для препятствий с особой геометрией (вроде Святой земли с ID 62, 63, 65 — явно прописано в коде).

Итог: почему всё всегда сходится

Три вещи работают вместе:

1. Фон — просто фон. Это статичная картинка, нарисованная художниками под известную сетку 17 × 11 с известными пиксельными координатами. Движок просто кладёт её в левый верхний угол.

2. Формула hexPositionLocal гарантирует, что все X ∈ [0..16] и Y ∈ [0..10] попадут в пиксели в диапазоне примерно (14, 86)…(784, 548) — внутри фона.

3. Алгоритм размещения препятствий ещё жёстче: отбраковывает все позиции, где любая блокируемая клетка вылезает за столбцы 3–13 или в первый/последний ряд. Поэтому «нарисованный» фон и «сгенерированные» препятствия сходятся в одном и том же визуально согласованном прямоугольнике.

По сути, весь боевой экран — это один большой ручной контракт, прибитый гвоздями в трёх местах: в формуле перевода гексов в пиксели, в ограничениях валидации препятствий и в самих растровых фонах, нарисованных 27 лет назад под эти же константы.

Ссылки на исходники

- `lib/ObstacleHandler.cpp` — шаблон препятствия и ключевая функция getBlocked
- `lib/battle/CObstacleInstance.cpp` — экземпляр препятствия на поле
- `lib/battle/BattleHex.h` — система координат и навигация по гексам
- `lib/battle/BattleInfo.cpp` — алгоритм размещения препятствий (строки 199–290)
- `lib/battle/CBattleInfoCallback.cpp` — обратный поиск: какое препятствие занимает гекс
- `client/battle/BattleFieldController.cpp` — фон, сетка, перевод гексов в пиксели
- `client/battle/BattleObstacleController.cpp` — отрисовка препятствий
- `config/obstacles.json` — каталог всех препятствий
- `config/battlefields.json` — соответствие типов поля и файлов фона

Репозиторий VCMI: https://github.com/vcmi/vcmi — все ссылки на строки относятся к ветке *develop* на момент апреля 2026 года.