

MINI DOOM

Современный C++ через игру



ТОМ 4

Здесь кто-то есть

C++20 · Visual Studio 2026

18+

8+

Алексей Примак

**MINI DOOM современный C++
через игру том 4 здесь кто то есть**

«Автор»

2026

Примак А.

MINI DOOM современный C++ через игру том 4 здесь кто то есть
/ А. Примак — «Автор», 2026

Четвёртый том серии «MINI DOOM. Современный C++ через игру»: в лабиринте появляются жители. Сначала аптечки и патроны — спрайты, которые честно прячутся за стенами благодаря Z-буферу. Потом имп и зомби, нарисованные кодом с помощью лямбд и шаблонов. Ты дашь игроку пистолет и дробовик с отдачей и разбросом, научишь монстров замечать тебя, гоняться по коридорам через конечный автомат и искать путь поиском в ширину, открывая двери. Имп бросит огненный шар, а ты узнаешь, почему быстрые снаряды пролетают сквозь стены и как это лечится. Финал тома — полноценный бой, в котором можно умереть. C++20, автотесты и настоящие скриншоты. 18+.

Содержание

Вместо предисловия	5
Серия целиком	6
Часть IV. Здесь кто-то есть	7
Глава 20. Спрайты	8
Предметы на карте	8
Картинка с дырками	9
Где спрайт на экране	12
Z-буфер: кто кого закрывает	15
Спрайты закрывают друг друга	18
Подбираем	18
Итоги главы	23
Упражнения	23
Задачи	25
Глава 21. Рисуем монстров кодом	26
Лямбды и шаблоны на пальцах	26
Кисти для спрайтов	29
Имп и зомби	34
Конец ознакомительного фрагмента.	37

Алексей Прима́к

MINI DOOM современный C++

через игру том 4 здесь кто то есть

Вместо предисловия

Это четвёртый том книги «MINI DOOM. Современный C++ через игру». К концу третьего тома у тебя трёхмерный лабиринт: лучи DDA, стены с текстурами без единого файла картинок, пол и потолок, затухание света и двери, которые открываются по E.

Теперь лабиринт перестаёт быть пустым. Номера глав, листингов и рисунков в серии сквозные, поэтому ссылка «см. главу 12» ведёт туда же, куда и в однотомнике.

Серия целиком

— **Том 1. Учимся держать оружие** — главы 1–8: основы C++ в консоли, от первой программы до памяти и санитайзеров.

— **Том 2. Окно в ад** — главы 9–13: своё окно Windows, пиксели, игровой цикл, клавиатура и мышь, текст.

— **Том 3. Третье измерение** — главы 14–19: рейкастинг, стены, текстуры, пол и свет, двери.

— **Том 4. Здесь кто-то есть** — главы 20–25: спрайты, монстры, оружие, искусственный интеллект, смерть.

— **Том 5. Шум, свет и финальные титры** — главы 26–29: звук, экраны, уровни, CMake и тесты.

Все исходники серии — на GitHub: <https://github.com/RushSex/MiniDoom>. Папка `examples/chNN` для каждой главы, итоговая игра — `examples/ch29/doom`. Там же README с инструкцией по сборке в Visual Studio 2026. Каждая глава начинается с проекта предыдущей, так что можно начать читать этот том, взяв готовый код последней главы прошлого тома.

Часть IV. Здесь кто-то есть

Лабиринт готов: стены, текстуры, свет, двери. Но бродить по пустому лабиринту — всё равно что ходить по ночному офису: красиво, жутковато и совершенно бессмысленно. Doom — это аптечки за углом, патроны в последний момент и, конечно, те, кто ждёт тебя в темноте.

В этой части мир оживёт. Сначала мы научимся ставить в него плоские картинки — спрайты — так, чтобы стены честно их закрывали. Потом нарисуем кодом монстров с анимацией, выдадим игроку пистолет и дробовик, научим монстров замечать игрока, гоняться за ним по коридорам и стрелять в ответ. В конце части в игре можно будет умереть — а значит, это уже игра.

Лирика. Ни один монстр Doom не был трёхмерным. Каждый имп — это набор плоских картинок, снятых с пластилиновых и латексных моделей с восьми сторон и отсканированных. Игра просто показывала нужную картинку, всегда повернутую лицом к игроку. Мы поступим так же, только картинки нарисуем кодом.

Глава 20. Спрайты

Спрайт — это плоская картинка, которая стоит в трёхмерном мире и всегда смотрит на игрока. Аптечка, коробка патронов, бочка, монстр — всё это спрайты. Чтобы нарисовать спрайт, нужно ответить на три вопроса: где он на экране, какого он размера и какие его части закрыты стенами. Первые два — это та же перспектива, что у стен. Третий — новая идея, **Z-буфер**.

К концу главы на уровне будут лежать аптечки и патроны, игрок будет их подбирать, а вверху экрана, как в Doom, появится надпись «PICKED UP A MEDIKIT».

Предметы на карте

Предметы, как и всё на уровне, рисуем прямо в карте: h — аптечка (health), а — патроны (ammo). Map::isWall считает стенами только цифры, поэтому буквы для движения — пустой пол. Сам предмет — маленькая структура, листинг 20.1.

```

items.hpp
1  // items.hpp — предметы, которые лежат на уровне
2  #pragma once
3
4  #include <cstdint>
5
6  namespace doom {
7
8  enum class ItemKind : std::uint8_t { Medkit, Clip };
9
10 struct Item {
11     ItemKind kind;
12     double x = 0.0;           // центр клетки, где лежит
13     double y = 0.0;
14     bool taken = false;
15 };
16
17 } // namespace doom

```

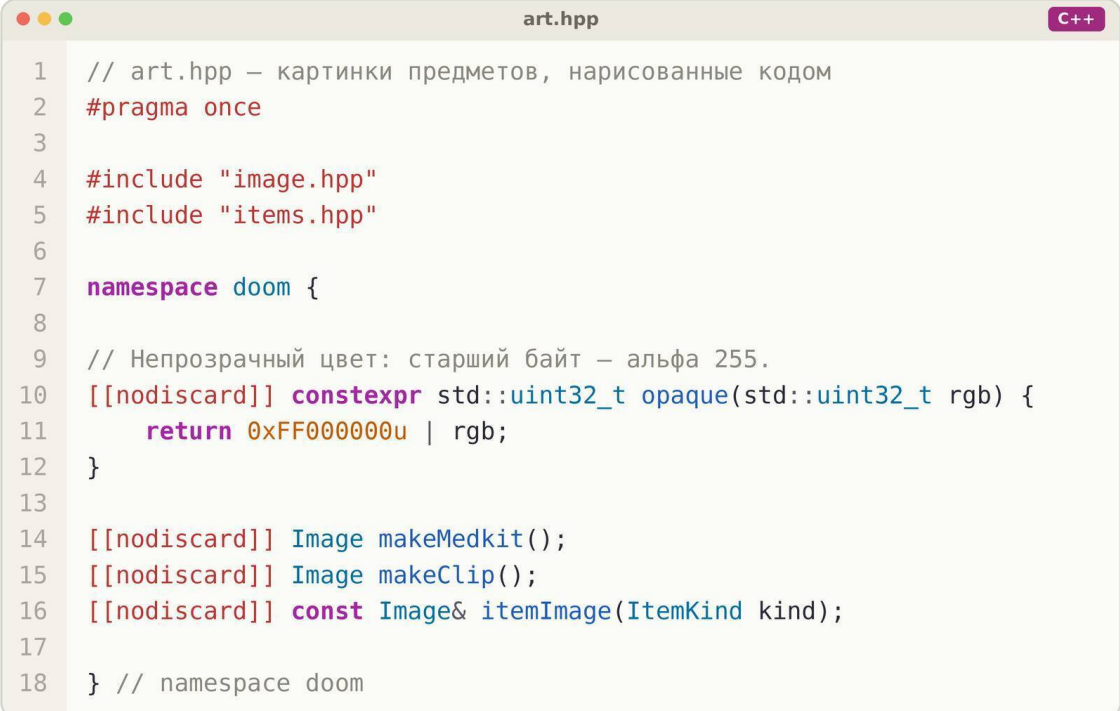
Листинг 20.1 — Предмет: что за вещь, где лежит, подобрана ли

enum class с указанием типа : std::uint8_t — перечисление занимает один байт, а не четыре, как int. Для двух значений это мелочь, но это привычка: в Doom предметов, монстров и состояний сотни.

В конструкторе Game пробегаем по карте и для каждой буквы h или a кладем предмет в центр клетки. Подобранный предмет не удаляется из вектора, а только помечается taken: так проще, а позже, в главе 25, по этому флагу мы вернём все предметы на место при перезапуске уровня.

Картинка с дырками

Стены — прямоугольники, их текстура заполнена целиком. А аптечка — не прямоугольник: вокруг неё должно просвечивать то, что сзади. Значит, у картинки спрайта должны быть **прозрачные** пиксели. Самый простой способ — занять под это старший байт цвета. Мы до сих пор писали цвет как `0x00RRGGBB`, и старший байт всегда был нулём. Теперь договоримся: в спрайтах этот байт — **альфа**, «непрозрачность». `0xFF` — пиксель есть, `0x00` — пикселя нет, сквозь него видно фон. Листинг 20.2 и 20.3.



```
1 // art.hpp — картинки предметов, нарисованные кодом
2 #pragma once
3
4 #include "image.hpp"
5 #include "items.hpp"
6
7 namespace doom {
8
9 // Непрозрачный цвет: старший байт — альфа 255.
10 [[nodiscard]] constexpr std::uint32_t opaque(std::uint32_t rgb) {
11     return 0xFF000000u | rgb;
12 }
13
14 [[nodiscard]] Image makeMedkit();
15 [[nodiscard]] Image makeClip();
16 [[nodiscard]] const Image& itemImage(ItemKind kind);
17
18 } // namespace doom
```

Листинг 20.2 — Непрозрачный цвет и картинки предметов

```
art.cpp C++
1 // art.cpp – аптечка и коробка патронов: прямоугольники с альфой
2 #include "art.hpp"
3
4 #include "canvas.hpp"
5
6 #include <array>
7
8 namespace doom {
9
10 namespace {
11 // Прямоугольник [x0, x1] × [y0, y1] на картинке спрайта.
12 void box(Image& img, int x0, int y0, int x1, int y1,
13         std::uint32_t c) {
14     for (int y = y0; y <= y1; ++y) {
15         for (int x = x0; x <= x1; ++x) {
16             img.set(x, y, opaque(c));
17         }
18     }
19 }
20 } // namespace
21
```

Листинг 20.3 — Аптечка и патроны из прямоугольников

```

art.cpp (продолжение)
22 Image makeMedkit() {
23     Image img(64, 64);           // всё прозрачное: нули
24     box(img, 20, 47, 43, 63, rgb(40, 40, 40)); // контур
25     box(img, 21, 48, 42, 62, rgb(225, 225, 220)); // коробка
26     box(img, 29, 50, 34, 60, rgb(210, 20, 20)); // красный
27     box(img, 25, 53, 38, 57, rgb(210, 20, 20)); // крест
28     return img;
29 }
30
31 Image makeClip() {
32     Image img(64, 64);
33     box(img, 21, 51, 42, 63, rgb(20, 18, 12));
34     box(img, 22, 52, 41, 62, rgb(80, 70, 50)); // коробка
35     for (int i = 0; i < 5; ++i) {           // пять патронов
36         const int x = 24 + i * 4;
37         box(img, x, 44, x + 2, 51, rgb(200, 160, 60));
38         box(img, x, 42, x + 2, 43, rgb(150, 100, 40));
39     }
40     return img;
41 }
42
43 const Image& itemImage(ItemKind kind) {
44     static const std::array<Image, 2> images{makeMedkit(),
45                                              makeClip()};
46     return images[static_cast<std::size_t>(kind)];
47 }
48
49 } // namespace doom

```

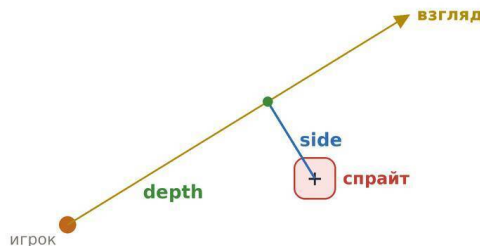
Листинг 20.3 (продолжение) — Аптечка и патроны из прямоугольников

Новая картинка Image заполнена нулями — то есть полностью прозрачна. Функция box закрашивает прямоугольник непрозрачным цветом. Аптечка — белая коробка с красным крестом в нижней части картинки 64 × 64: спрайт, как и стена, занимает одну клетку в высоту, а маленький предмет должен лежать на полу, а не висеть посередине.

Запомни. Цвет 0xAARRGGBB: альфа — старший байт. Для стен и холста она не используется и равна нулю, для спрайтов 0xFF — непрозрачно, 0x00 — прозрачно. Функция shade из главы 18 работает только с масками 0xFF00FF и 0x00FF00, поэтому альфу она просто отбрасывает, и на холст попадает обычный цвет.

Где спрайт на экране

Спрайт стоит в точке (sx, sy) . Чтобы понять, куда его рисовать, переведём эту точку в **систему координат камеры**: сколько до неё вперёд по взгляду и сколько вбок, вправо. Вперёд — это проекция вектора «от игрока к спрайту» на направление взгляда, а вбок — на направление «вправо» из главы 14, рисунок 20.1.



```
depth = (s - игрок) · dir
side = (s - игрок) · право
```

На экране:

```
x = w/2 · (1 + side / (0,8 · depth))
размер = h / depth
низ = h/2 + (h/2) / depth
```

Рисунок 20.1 — Спрайт в системе камеры: *depth* — вперёд по взгляду, *side* — вбок

Дальше всё уже знакомо. *depth* — это ровно то расстояние вдоль взгляда, которое мы мерили для стен в главе 16, поэтому и размер спрайта считается так же: h / depth . Положение по горизонтали — из окна камеры: окно на расстоянии 1 имеет полуширину 0,8, на расстоянии *depth* — $0,8 \times \text{depth}$, и спрайт, сдвинутый вбок на *side*, попадает в точку $\text{side} / (0,8 \times \text{depth})$ от -1 до $+1$. Низ спрайта — на полу, а пол на расстоянии *depth* виден на $(h/2) / \text{depth}$ ниже горизонта, это формула из главы 18.

Вся отрисовка спрайтов — функция `drawSprites`, листинги 20.4 и 20.5.

```

sprites.hpp
1  // sprites.hpp – плоские картинки в трёхмерном мире
2  #pragma once
3
4  #include "canvas.hpp"
5  #include "image.hpp"
6
7  #include <span>
8  #include <vector>
9
10 namespace doom {
11
12 struct Sprite {
13     double x = 0.0;           // где стоит на карте
14     double y = 0.0;
15     const Image* image = nullptr;
16     double depth = 0.0;      // расстояние вдоль взгляда
17 };
18
19 struct Camera {
20     double x = 0.0;           // где стоит игрок
21     double y = 0.0;
22     double angle = 0.0;
23 };
24
25 // zbuf – расстояние до стены в каждом столбце экрана.
26 void drawSprites(Canvas& canvas, std::vector<Sprite> sprites,
27                 const Camera& cam, std::span<const double> zbuf);
28
29 } // namespace doom

```

Листинг 20.4 — Спрайт и камера

```
sprites.cpp C++
1 // sprites.cpp – проекция, сортировка и Z-буфер
2 #include "sprites.hpp"
3
4 #include "shade.hpp"
5 #include "view3d.hpp"
6
7 #include <algorithm>
8 #include <cmath>
9 #include <functional>
10
11 namespace doom {
12
13 void drawSprites(Canvas& canvas, std::vector<Sprite> sprites,
14                  const Camera& cam, std::span<const double> zbuf) {
15     const double w = canvas.width();
16     const double h = canvas.height();
17     const double dirX = std::cos(cam.angle);
18     const double dirY = std::sin(cam.angle);
19     for (Sprite& s : sprites) { // насколько далеко впереди
20         s.depth = (s.x - cam.x) * dirX + (s.y - cam.y) * dirY;
21     }
22     // Художник: сначала дальние, ближние рисуются поверх.
23     std::ranges::sort(sprites, std::greater{}, &Sprite::depth);
24
25     for (const Sprite& s : sprites) {
26         const double d = s.depth;
27         if (d < 0.1) {
28             continue; // позади или в упор
29         }
30         // Насколько правее взгляда (в клетках) – и куда на экран.
31         const double rx = s.x - cam.x;
32         const double ry = s.y - cam.y;
```

Листинг 20.5 — *drawSprites*: проекция, сортировка, Z-буфер, прозрачность

```

sprites.cpp (продолжение)
33     const double side = ry * dirX - rx * dirY;
34     const double mid = w / 2 * (1 + side / (kPlane * d));
35     const double size = h / d;      // клетка на таком удалении
36     const double left = mid - size / 2;
37     const double top = h / 2 + h / 2 / d - size; // низ — на полу
38     const double step = s.image->width() / size;
39     const std::uint32_t light = lightAt(d);
40
41     // Прямоугольник спрайта на экране, обрезанный по краям.
42     const auto px = [](double v) { return static_cast<int>(v); };
43     const int x0 = std::max(0, px(std::ceil(left)));
44     const int x1 = std::min(px(w), px(left + size));
45     const int y0 = std::max(0, px(std::ceil(top)));
46     const int y1 = std::min(px(h), px(top + size));
47     for (int col = x0; col < x1; ++col) {
48         if (d >= zbuf[static_cast<std::size_t>(col)]) {
49             continue;          // столбец закрыт стеной
50         }
51         const int tx = px((col - left) * step) & 63;
52         for (int row = y0; row < y1; ++row) {
53             const int ty = px((row - top) * step) & 63;
54             const std::uint32_t c = s.image->at(tx, ty);
55             if ((c >> 24) != 0) { // альфа 0 — прозрачно
56                 canvas.put(col, row, shade(c, light));
57             }
58         }
59     }
60 }
61 }
62
63 } // namespace doom

```

Листинг 20.5 (продолжение) — *drawSprites*: проекция, сортировка, Z-буфер, прозрачность

Строки 19–21 — *depth* для каждого спрайта. **Строки 31–39** — формулы с рисунка 20.1: *side*, середина на экране, размер, левый и верхний края, шаг по картинке и свет по расстоянию, как у стен.

Строки 41–46 — прямоугольник спрайта на экране, обрезанный по краям. Лямбда *px* — просто короткое имя для *static_cast<int>*, чтобы строки помещались на экран.

Строки 47–59 — рисование. Внешний цикл по столбцам, внутренний по строкам. Пиксель картинки рисуем, только если его альфа не ноль. А на **строке 48** — главное в этой главе.

Z-буфер: кто кого закрывает

Стены мы рисуем раньше спрайтов. Если просто нарисовать спрайт поверх, он окажется виден сквозь стены — аптечка из соседней комнаты будет висеть на кирпичках. Нужно знать, где стена ближе спрайта, а где дальше. И эта информация у нас уже есть: для каждого столбца

экрана мы знаем расстояние до стены. Раньше мы его выбрасывали, а теперь drawView записывает его в массив, листинг 20.6.

```

view3d.hpp
1 // view3d.hpp — что видит игрок: стены в перспективе
2 #pragma once
3
4 #include "canvas.hpp"
5 #include "map.hpp"
6
7 #include <span>
8
9 namespace doom {
10
11 inline constexpr double kPlane = 0.8; // полуширина окна камеры
12
13 // Рисует пол, потолок и стены. В zbuf записывает расстояние
14 // до стены в каждом столбце — оно понадобится спрайтам.
15 void drawView(Canvas& canvas, const Map& map, double x, double y,
16             double angle, std::span<double> zbuf);
17
18 } // namespace doom

```

Листинг 20.6 — drawView теперь заполняет zbuf

Этот массив — по одному числу на каждый из 640 столбцов — и называется **Z-буфером**, буфером глубины: «z» по традиции обозначает глубину сцены. Спрайт в столбце col рисуется, только если он ближе стены в этом столбце: $depth < zbuf[col]$. Рисунок 20.2.



Рисунок 20.2 — Z-буфер: спрайт на глубине 3 рисуется только в столбцах, где стена дальше

Посмотрим, что будет без этой проверки. Уберём строки 48–50 из листинга 20.5 — листинг 20.7 — и снимем кадр со стартовой позиции, рисунок 20.3.

```

main.cpp
47     for (int col = x0; col < x1; ++col) {
48         // ОШИБКА: забыли проверить zbuf — стены не закрывают.
49         const int tx = px((col - left) * step) & 63;
50         for (int row = y0; row < y1; ++row) {
51             const int ty = px((row - top) * step) & 63;
52             const std::uint32_t c = s.image->at(tx, ty);

```

Листинг 20.7 — Ошибка: Z-буфер не проверяется



Рисунок 20.3 — Без Z-буфера: на кирпичной стене висят прозрачные патроны и аптечка из соседней комнаты

Присмотрись к кирпичной стене и к металлической колонне справа: на них нарисованы тёмные силуэты — коробка патронов и аптечка из соседней комнаты. Они далеко, поэтому маленькие и тёмные, но они есть и просвечивают сквозь стену. С проверкой их нет.

Грабли. Z-буфер — по столбцам, а не по пикселям. Это работает, потому что у нас все стены от пола до потолка: если стена в столбце есть, она закрывает его целиком. В Doom у стен бывают разные высоты, ступеньки и окна, и там приходилось хранить для каждого столбца верхнюю и нижнюю границу видимого. А видеокарты хранят глубину для **каждого пикселя** — и это тоже называется Z-буфером.

Спрайты закрывают друг друга

Стены от спрайтов отгородили. Но спрайты могут перекрывать друг друга: аптечка перед коробкой патронов. Z-буфер тут не поможет, в нём только стены. Поможет старый «алгоритм художника» из главы 18: рисуем от дальних к ближним, и ближние ложатся поверх. Для этого спрайты сортируются по depth по убыванию — **строка 23** листинга 20.5:

```
std::ranges::sort(sprites, std::greater{ }, &Sprite::depth);
```

Это сортировка из C++20 с двумя дополнительными аргументами. Второй — **сравнение**: `std::greater{ }` — «больше», значит, по убыванию. Третий — **проекция**: указатель на поле `&Sprite::depth` говорит, что сравнивать надо не спрайты целиком, а только их глубины. Без проекции пришлось бы писать лямбду `[] (const Sprite& a, const Sprite& b) { return a.depth > b.depth; }` — то же самое, только длиннее и с шансом перепутать знак.

Обрати внимание: `drawSprites` получает вектор **по значению**, то есть копию. Сортировка переставляет копию, а исходный порядок у вызывающего не меняется. Копия пары десятков спрайтов ничего не стоит, а функция становится честнее: она ничего не портит снаружи.

Подбираем

Остаётся подбирать предметы. Каждый кадр смотрим, не подошёл ли игрок к предмету ближе чем на полклетки, листинг 20.8.

```
game.cpp C++
159 void Game::pickUpItems() {
160     for (Item& item : items_) {
161         const double dx = item.x - x_;
162         const double dy = item.y - y_;
163         if (item.taken || dx * dx + dy * dy > kPickup * kPickup) {
164             continue;
165         }
166         if (item.kind == ItemKind::Medkit) {
167             if (health_ >= 100) {
168                 continue;           // здоров – аптечка лежит дальше
169             }
170             health_ = std::min(100, health_ + 25);
171             message_ = "PICKED UP A MEDIKIT";
172         } else {
173             ammo_ += 10;
174             message_ = "PICKED UP A CLIP";
175         }
176         item.taken = true;
177         messageTime_ = kMessage;
178     }
179 }
180
```

Листинг 20.8 — Подбор предметов и сборка спрайтов для кадра

```

game.cpp (продолжение)
181 void Game::render(Canvas& canvas) const {
182     if (showMap_) {
183         drawTopView(canvas, map_, x_, y_, angle_);
184         return;
185     }
186     const auto w = static_cast<std::size_t>(canvas.width());
187     std::vector<double> zbuf(w); // расстояние до стены в столбце
188     drawView(canvas, map_, x_, y_, angle_, zbuf);
189     std::vector<Sprite> sprites;
190     for (const Item& item : items_) {
191         if (!item.taken) {
192             const Image* img = &ItemImage(item.kind);
193             sprites.push_back({item.x, item.y, img});
194         }
195     }
196     drawSprites(canvas, std::move(sprites), {x_, y_, angle_}, zbuf);
197 }

```

Листинг 20.8 (продолжение) — Подбор предметов и сборка спрайтов для кадра

Строки 161–165. Расстояние сравниваем в квадрате: $dx^2 + dy^2 > 0,5^2$. Корень не нужен — если расстояние больше, то и его квадрат больше. Мелкая, но приятная экономия: эту проверку мы будем делать для каждого предмета и монстра каждый кадр.

Строки 166–169 — правило из Doom: если здоровье уже 100%, аптечка остаётся лежать. Её можно вернуться и взять потом, когда понадобится. Патроны берутся всегда.

Строки 181–197 — сборка кадра. Z-буфер — локальный вектор: `render` константный и не может менять поля `Game`, да и не нужно — буфер живёт один кадр. Для каждого неподобранного предмета кладём спрайт с указателем на его картинку. Картинки живут в `static`-массиве `ItemImage` до конца программы, поэтому указатель на них безопасен.

HUD теперь показывает здоровье, патроны и сообщение — по примеру Doom, оно висит вверху слева две секунды. Робот проверяет — листинг 20.9.

```

1  // test.cpp — подбираем предметы; кадр со спрайтами
2  #include "game.hpp"
3  #include "hud.hpp"
4  #include "level.hpp"
5
6  #include <algorithm>
7  #include <format>
8  #include <fstream>
9  #include <iostream>
10 #include <string_view>
11
12 namespace {
13 void hold(doom::Game& game, const doom::Input& in, double seconds) {
14     const int frames = static_cast<int>(seconds * 60.0 + 0.5);
15     for (int i = 0; i < frames; ++i) {
16         game.update(in, 1.0 / 60.0);
17     }
18 }
19
20 void report(const doom::Game& g, std::string_view what) {
21     const auto left = std::ranges::count(g.items(), false,
22                                         &doom::Item::taken);
23     std::cout << std::format("{:<14} x {:.2f} hp {:3} ammo {:2} "
24                             "left {} \n{}\n",
25                             what, g.playerX(), g.health(), g.ammo(),
26                             left, g.message());
27 }
28

```

Листинг 20.9 — Автотест: идём вдоль комнаты и подбираем

```

test.cpp (продолжение) C++
29 void savePPM(const doom::Canvas& c, const char* path) {
30     std::ofstream f(path, std::ios::binary);
31     f << "P6\n" << c.width() << ' ' << c.height() << "\n255\n";
32     for (int y = 0; y < c.height(); ++y) {
33         for (int x = 0; x < c.width(); ++x) {
34             const std::uint32_t p = c.at(x, y);
35             f.put(static_cast<char>((p >> 16) & 0xFF));
36             f.put(static_cast<char>((p >> 8) & 0xFF));
37             f.put(static_cast<char>(p & 0xFF));
38         }
39     }
40 }
41 } // namespace
42
43 int main() {
44     doom::Game game{doom::Map{doom::kLevel}};
45     report(game, "start");
46
47     doom::Canvas canvas(640, 400);
48     game.render(canvas);
49     doom::drawHud(canvas, game, 60);
50     savePPM(canvas, "frame.ppm");
51
52     hold(game, {.forward = true}, 0.9);
53     report(game, "forward 0.9 s");
54     hold(game, {.forward = true}, 0.7);
55     report(game, "forward 0.7 s");
56     hold(game, {}, 2.0);
57     report(game, "waited 2 s");
58 }

$ ./a.out
start      x 1.50  hp  80  ammo 20  left 8  ""
forward 0.9 s  x 4.20  hp 100  ammo 20  left 7  "PICKED UP A MEDIKIT"
forward 0.7 s  x 6.30  hp 100  ammo 20  left 7  "PICKED UP A MEDIKIT"
waited 2 s    x 6.30  hp 100  ammo 20  left 7  ""

```

Листинг 20.9 (продолжение) — Автотест: идём вдоль комнаты и подбираем

— **start** — на уровне 8 предметов, здоровье 80%.

— **forward 0.9 s** — игрок прошёл через первую аптечку: здоровье $80 + 25 = 105$, но не больше 100. Предметов осталось 7, сообщение висит.

— **forward 0.7 s** — прошли через вторую аптечку, но здоровье 100%, и она осталась лежать: всё ещё 7. Сообщение — от первой, оно висит две секунды.

— **waited 2 s** — сообщение исчезло.



Рисунок 20.4 — Кадр со старта: две аптечки, ближняя закрывает дальнюю

На рисунке 20.4 обе аптечки — и ближняя честно закрывает дальнюю. В окне робот делает то же: идёт вперёд и подбирает аптечку — рисунок 20.5.



Рисунок 20.5 — Контрольная точка: аптечка подобрана, сообщение сверху, здоровье 100%

По-взрослому. Мы рисуем спрайт пиксель за пикселем с проверкой альфы. В Doom так же, только картинки хранились не прямоугольником, а столбцами «отрезков»: для каждого столбца записано, с какой строки начинается непрозрачная часть и сколько в ней пикселей. Прозрачные куски не хранились вовсе и не перебирались при рисовании. Этот формат назывался `ratch`, и в нём лежат все монстры и предметы Doom.

Итоги главы

- Спрайт — плоская картинка, всегда повёрнутая к игроку.
- Позиция в системе камеры: `depth` — проекция на взгляд, `side` — проекция на «вправо».
- На экране: $x = w/2 \times (1 + side / (0,8 \times depth))$, размер $h / depth$, низ на полу.
- Прозрачность — альфа в старшем байте цвета: 0 — пикселя нет.
- Z-буфер — расстояние до стены в каждом столбце; спрайт рисуется, где он ближе стены.
- Спрайты между собой — от дальних к ближним: `std::ranges::sort(v, std::greater{ }, &Sprite::depth)`.

Контрольная точка: аптечки и патроны лежат в мире и подбираются. Но предметы скучные — коробки из прямоугольников. В следующей главе мы научимся рисовать кодом настоящих монстров: с объёмом, контуром и анимацией.

Упражнения

Упражнение 20.1. *Что выведет?* Листинг 20.10 сортирует предметы по глубине.

```
main.cpp C++
1  #include <algorithm>
2  #include <functional>
3  #include <iostream>
4  #include <string>
5  #include <vector>
6
7  struct Thing {
8      std::string name;
9      double depth;
10 };
11
12 int main() {
13     std::vector<Thing> things{{"medkit", 3.0}, {"clip", 7.5},
14                               {"barrel", 1.2}, {"imp", 4.0}};
15     std::ranges::sort(things, std::greater{}, &Thing::depth);
16     for (const Thing& t : things) {
17         std::cout << t.name << ' ';
18     }
19     std::cout << '\n';
20 }
```

Листинг 20.10 — Что выведет программа?

Упражнение 20.2. *Найди баг.* В листинге 20.11 программист сортирует спрайты и рисует их по порядку. Ближний спрайт А на экране оказывается закрыт дальними. Почему?



```

1  #include <algorithm>
2  #include <iostream>
3  #include <vector>
4
5  struct Sprite {
6      char id;
7      double depth;
8  };
9
10 int main() {
11     std::vector<Sprite> sprites{{'A', 2.0}, {'B', 6.0}, {'C', 4.0}};
12     std::ranges::sort(sprites, {}, &Sprite::depth);
13     std::cout << "draw order:";
14     for (const Sprite& s : sprites) { // рисуем по очереди
15         std::cout << ' ' << s.id << '(' << s.depth << ')';
16     }
17     std::cout << '\n';
18 }

```

\$./a.out
draw order: A(2) C(4) B(6)

Листинг 20.11 — Порядок рисования спрайтов

Упражнение 20.3. *Напиши сам.* Добавь на карту бочку — букву b. Бочку нельзя подод-
брать, она просто стоит. Нарисуй её кодом: зелёный цилиндр с тёмными обручами.

Упражнение 20.4. *Подумай.* Почему у спрайта и у стены на одном расстоянии одина-
ковая высота на экране? Как сделать маленький спрайт, например гильзу, если размер всегда
 h / depth ?

Задачи

Задача 20.1. Сообщения по очереди. Если подобрать два предмета подряд за полсе-
кунды, второе сообщение затирает первое. Сделай очередь сообщений: каждое показывается
хотя бы секунду.

Задача 20.2. Бочки мешают проходить. Сделай, чтобы бочка из упражнения 20.3 была
препятствием, как стена. Подумай: бочка — круглая, а стены — клетки.

Задача 20.3. Свечение. В Doom некоторые предметы светятся сами и не темнеют вдали.
Добавь в Sprite флаг `fullbright` и для таких спрайтов используй яркость 256 вместо `lightAt`.

Глава 21. Рисуем монстров кодом

Аптечку можно собрать из трёх прямоугольников. С монстром так не выйдет: прямоугольный имп выглядит как шкаф с рогами. Живому существу нужны округлые формы, объём, контур — то, что художник делает кистью. Мы будем рисовать кодом, и наша кисть — функции: эллипс с объёмом, многоугольник, обводка.

Попутно разберём две вещи из C++, без которых такие кисти пишутся тяжело. **Лямбды** — маленькие функции прямо внутри кода, которые могут видеть переменные вокруг себя. И **шаблоны функций** — функции, которые работают с любым подходящим типом. Мы ими уже пользовались, не называя: `std::ranges::sort` из главы 20 — шаблон, а `[](int x, int y) { ... }` в главе 14 — лямбда. Теперь разберёмся по-честному.

Лямбды и шаблоны на пальцах

Почти каждая кисть устроена одинаково: пройти по всем пикселям картинки и для каждого решить, красить ли его и каким цветом. Обход один и тот же, отличается только «решение». Значит, обход можно написать один раз, а решение передавать внутрь. Посмотрим на этот приём в консоли — листинг 21.1.

```

1  #include <cstdlib>
2  #include <iostream>
3  #include <string>
4  #include <vector>
5
6  // Шаблон функции: F – любой тип, который можно вызвать как f(x, y).
7  template <typename F>
8  void forEachCell(int w, int h, F f) {
9      for (int y = 0; y < h; ++y) {
10         for (int x = 0; x < w; ++x) {
11             f(x, y);
12         }
13     }
14 }
15
16 int main() {
17     const int w = 36;
18     const int h = 13;
19     std::vector<std::string> pic(h, std::string(w, '.'));
20     // Лямбда захватывает pic по ссылке и прячет приведения типов.
21     const auto put = [&pic](int x, int y, char c) {
22         const auto row = static_cast<std::size_t>(y);
23         pic[row][static_cast<std::size_t>(x)] = c;
24     };
25     // Эллипс с центром (10, 6) и полуосями 9 и 6, «с объёмом».
26     const double cx = 10.0, cy = 6.0, rx = 9.0, ry = 6.0;
27     forEachCell(w, h, [&](int x, int y) {
28         const double nx = (x - cx) / rx;
29         const double ny = (y - cy) / ry;
30         const double d = nx * nx + ny * ny;

```

Листинг 21.1 — Эллипс и треугольник буквами: шаблон обхода и лямбды

```

31         if (d <= 1.0) {
32             const double light = 1.0 - 0.4 * (nx + ny) - 0.3 * d;
33             put(x, y, light > 1.0 ? '@' : light > 0.7 ? '#' : '+');
34         }
35     });
36     // Треугольник: точка внутри, если она по одну сторону
37     // от всех трёх рёбер.
38     struct P {
39         int x;
40         int y;
41     };
42     const P a{22, 11}, b{29, 1}, q{35, 11};
43     const auto cross = [](P p0, P p1, int x, int y) {
44         return (p1.x - p0.x) * (y - p0.y) -
45             (p1.y - p0.y) * (x - p0.x);
46     };
47     forEachCell(w, h, [&](int x, int y) {
48         const int c1 = cross(a, b, x, y);
49         const int c2 = cross(b, q, x, y);
50         const int c3 = cross(q, a, x, y);
51         const bool neg = c1 < 0 || c2 < 0 || c3 < 0;
52         const bool pos = c1 > 0 || c2 > 0 || c3 > 0;
53         if (!(neg && pos)) {
54             put(x, y, '^');
55         }
56     });
57     for (const std::string& row : pic) {
58         std::cout << row << '\n';
59     }
60 }

```

Листинг 21.1 (продолжение) — Эллипс и треугольник буквами: шаблон обхода и лямбды

Строки 6–14 — шаблон функции. Слово `template <typename F>` говорит: `F` — какой-то тип, какой — решим, когда функцию вызовут. Внутри с `f` обращаются как с функцией: `f(x, y)`. Подойдёт что угодно, что можно так вызвать. При каждом вызове с новым типом компилятор сам напишет отдельную версию `forEachCell` под этот тип — поэтому шаблон работает так же быстро, как если бы цикл был написан вручную.

Строки 21–24 — лямбда. Квадратные скобки — **список захвата**: что лямбда берёт из окружения. `[&ric]` — взять переменную `ric` по ссылке, чтобы лямбда могла её менять. Круглые скобки — параметры, как у обычной функции. Тело — в фигурных. Лямбду можно сохранить в переменную (её тип знает только компилятор, поэтому пишем `auto`) и вызывать как функцию: `put(x, y, '@')`.

Строки 27–35 — лямбда с захватом `[&]`: «всё, что упомянуто, — по ссылке». Она видит `sx`, `sy`, `gx`, `gy` и `put`. Внутри — формула эллипса: точка внутри, если $(x - sx)^2 / gx^2 + (y - sy)^2 / gy^2 \leq 1$. И сразу «объём»: чем ближе к левому верхнему краю, тем светлее, к краям — темнее. Буквы `@`, `#` и `+` — три уровня яркости.

Строки 36–56 — треугольник. Как понять, что точка внутри? Возьмём ребро `AB` и посмотрим, с какой стороны от него лежит точка. Это говорит знак **векторного произведения** — формула в лямбде `cross`. Внутри выпуклой фигуры точка лежит с **одной и той же** стороны от всех рёбер, обойдённых по кругу. Если для одних рёбер знак плюс, а для других минус — точка снаружи. Рисунок 21.1.

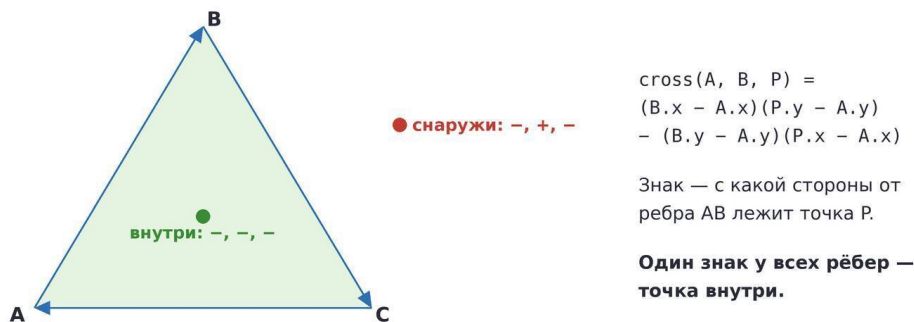


Рисунок 21.1 — Точка внутри выпуклого многоугольника — по одну сторону от всех рёбер

Грабли. Захват по ссылке `[&]` удобен, но опасен: лямбда хранит не значения, а ссылки на переменные. Пока переменные живы — всё хорошо. Но если сохранить такую лямбду и вызвать её после выхода из функции, где жили переменные, ссылки будут висячими — та самая мёртвая память из главы 8. Правило: `[&]` — для лямбд, которые вызываются тут же, как в `forEachCell`. Если лямбду сохраняют надолго — захватывай по значению.

Кисти для спрайтов

Те же идеи, только на настоящей картинке `Image` с альфой. Набор кистей — листинги 21.2 и 21.3.

```

1  // paint.hpp — кисти для рисования спрайтов кодом
2  #pragma once
3
4  #include "image.hpp"
5
6  #include <cstdint>
7  #include <span>
8
9  namespace doom {
10
11  struct Point {
12      double x = 0.0;
13      double y = 0.0;
14  };
15
16  // Вызывает f(x, y) для каждого пикселя картинки. f — что угодно,
17  // что можно вызвать с двумя int: функция, лямбда, объект.
18  template <typename F>
19  void forEachPixel(Image& img, F f) {
20      for (int y = 0; y < img.height(); ++y) {
21          for (int x = 0; x < img.width(); ++x) {
22              f(x, y);
23          }
24      }
25  }
26

```

Листинг 21.2 — Кисти: обход, эллипс, многоугольник, обводка, подкраска

```

27  // Цвет, умноженный на яркость k (1 — как есть), непрозрачный.
28  [[nodiscard]] std::uint32_t lit(std::uint32_t base, double k);
29
30  // Эллипс с «объёмом»: свет падает слева сверху.
31  void fillEllipse(Image& img, Point c, double rx, double ry,
32                  std::uint32_t base);
33  // Выпуклый многоугольник: вершины по кругу в любую сторону.
34  void fillPolygon(Image& img, std::span<const Point> pts,
35                  std::uint32_t base);
36  // Тёмная обводка вокруг всех непрозрачных пикселей.
37  void outline(Image& img, std::uint32_t base);
38  // Подмешать цвет ко всей фигуре: amount от 0 до 1.
39  void tint(Image& img, std::uint32_t base, double amount);
40
41  } // namespace doom

```

Листинг 21.2 (продолжение) — Кисти: обход, эллипс, многоугольник, обводка, подкраска

`forEachPixel` — тот же шаблон, что в листинге 21.1, но для `Image`. Шаблоны пишут целиком в заголовке: компилятор должен видеть их тело в каждом файле, где их вызывают, чтобы сделать нужную версию. Остальные функции — обычные, их тела в `paint.cpp`.

```

1  // paint.cpp – эллипс, многоугольник, обводка, подкраска
2  #include "paint.hpp"
3
4  #include "art.hpp"
5  #include "canvas.hpp"
6
7  #include <algorithm>
8
9  namespace doom {
10
11  std::uint32_t lit(std::uint32_t base, double k) {
12      const auto ch = [k](std::uint32_t v) {
13          const double s = std::clamp(v * k, 0.0, 255.0);
14          return static_cast<std::uint32_t>(s);
15      };
16      const std::uint32_t r = ch((base >> 16) & 0xFF);
17      const std::uint32_t g = ch((base >> 8) & 0xFF);
18      const std::uint32_t b = ch(base & 0xFF);
19      return opaque(rgb(r, g, b));
20  }
21
22  void fillEllipse(Image& img, Point c, double rx, double ry,
23                  std::uint32_t base) {
24      forEachPixel(img, [&](int x, int y) {
25          // Координаты внутри эллипса: от -1 до 1 по каждой оси.
26          const double nx = (x + 0.5 - c.x) / rx;
27          const double ny = (y + 0.5 - c.y) / ry;
28          const double d = nx * nx + ny * ny;
29          if (d > 1.0) {
30              return; // снаружи
31          }

```

Листинг 21.3 — Кисти: реализация

```
paint.cpp (продолжение) C++
32     // Слева сверху светлее, к краям темнее — шар, а не блин.
33     const double light = 1.1 - 0.3 * (nx + ny) - 0.3 * d;
34     img.set(x, y, lit(base, light));
35 });
36 }
37
38 void fillPolygon(Image& img, std::span<const Point> pts,
39                 std::uint32_t base) {
40     forEachPixel(img, [&](int x, int y) {
41         const Point p{x + 0.5, y + 0.5};
42         bool left = false;           // точка слева от ребра
43         bool right = false;          // точка справа от ребра
44         for (std::size_t i = 0; i < pts.size(); ++i) {
45             const Point a = pts[i];
46             const Point b = pts[(i + 1) % pts.size()];
47             // Векторное произведение: знак говорит, с какой стороны.
48             const double cross = (b.x - a.x) * (p.y - a.y) -
49                                 (b.y - a.y) * (p.x - a.x);
50             left = left || cross < 0.0;
51             right = right || cross > 0.0;
52         }
53         if (!(left && right)) {       // со всех сторон одинаково
54             img.set(x, y, lit(base, 1.0));
55         }
56     });
57 }
58
```

Листинг 21.3 (продолжение) — Кисти: реализация

```

59 void outline(Image& img, std::uint32_t base) {
60     const Image src = img;           // смотрим на копию «до»
61     const auto solid = [&src](int x, int y) {
62         return x >= 0 && y >= 0 && x < src.width() &&
63             y < src.height() && (src.at(x, y) >> 24) != 0;
64     };
65     forEachPixel(img, [&](int x, int y) {
66         if (solid(x, y)) {
67             return;
68         }
69         if (solid(x - 1, y) || solid(x + 1, y) || solid(x, y - 1) ||
70             solid(x, y + 1)) {
71             img.set(x, y, opaque(base));
72         }
73     });
74 }
75
76 void tint(Image& img, std::uint32_t base, double amount) {
77     forEachPixel(img, [&](int x, int y) {
78         const std::uint32_t c = img.at(x, y);
79         if ((c >> 24) == 0) {
80             return;
81         }
82         const auto mix = [&](int shift) {
83             const double a = (c >> shift) & 0xFF;
84             const double b = (base >> shift) & 0xFF;
85             return static_cast<std::uint32_t>(a + (b - a) * amount);
86         };
87         img.set(x, y, opaque(rgb(mix(16), mix(8), mix(0))));
88     });
89 }
90
91 } // namespace doom

```

Листинг 21.3 (продолжение) — Кисти: реализация

lit (строки 11–20) умножает каждый канал цвета на яркость *k* и зажимает в 0...255 — так, в отличие от *shade* из главы 18, можно и осветлять, *k* больше единицы. Внутренняя лямбда *ch* захватывает *k* по значению: [*k*] — ей нужно только прочесть число.

fillEllipse (строки 22–36) — эллипс из листинга 21.1, только вместо трёх букв — плавная яркость.

fillPolygon (строки 38–57) — проверка знаков векторного произведения для любого числа вершин. Вершины передаются через `std::span<const Point>` — «окно» на массив точек, как в главе 14 у *Map*. Подойдёт `std::array`, `std::vector` или обычный массив. Ребро от последней вершины к первой даёт $(i + 1) \% \text{size}$: после последнего индекса — снова ноль.

outline (строки 59–74) обводит фигуру: прозрачный пиксель, рядом с которым есть непрозрачный, красится тёмным. Важная деталь — строка 60: смотрим мы на копию картинки, сделанную до обводки, а пишем в оригинал. Почему это важно, покажет упражнение 21.2.

tint (строки 76–89) подмешивает цвет ко всей фигуре — пригодится, когда монстр получит пулю и покраснеет.

Имп и зомби

Теперь можно рисовать. Имп и зомби — листинг 21.4. Каждый — пара десятков вызовов кистей, слой за слоем: сначала то, что сзади, потом то, что спереди. Ноги, туловище, руки, голова, глаза, обводка.

```

1 // monsters.cpp — имп и зомби из эллипсов и многоугольников
2 #include "monsters.hpp"
3
4 #include "canvas.hpp"
5 #include "paint.hpp"
6
7 #include <array>
8 #include <cstdint>
9
10 namespace doom {
11
12     namespace {
13         // Прямоугольник — частный случай многоугольника.
14         void rect(Image& img, double x0, double y0, double x1, double y1,
15                 std::uint32_t c) {
16             const std::array<Point, 4> p{{{x0, y0}, {x1, y0}, {x1, y1},
17                                         {x0, y1}}};
18             fillPolygon(img, p, c);
19         }
20
21         void tri(Image& img, Point a, Point b, Point c, std::uint32_t col) {
22             const std::array<Point, 3> p{a, b, c};
23             fillPolygon(img, p, col);
24         }
25
26         // Шаг ноги: в двух позах ноги сдвинуты в разные стороны.
27         double stepOf(Pose pose) {
28             return pose == Pose::Walk1 ? -3.0 : 3.0;
29         }
30     } // namespace
31

```

Листинг 21.4 — Монстры из эллипсов и многоугольников

```

monsters.cpp (продолжение)
32 Image makeImp(Pose pose) {
33     Image img(64, 64);
34     const auto skin = rgb(150, 95, 55);
35     const auto dark = rgb(95, 55, 30);
36     const auto bone = rgb(225, 215, 185);
37     const double s = stepOf(pose);
38     rect(img, 24 + s, 46, 29 + s, 63, dark);           // ноги
39     rect(img, 35 - s, 46, 40 - s, 63, dark);
40     fillEllipse(img, {32, 37}, 11, 12, skin);           // туловище
41     fillEllipse(img, {32, 34}, 6, 5, rgb(175, 125, 80)); // грудь
42     fillEllipse(img, {20, 40 - s / 2}, 3.5, 9, skin);   // руки
43     fillEllipse(img, {44, 40 + s / 2}, 3.5, 9, skin);
44     tri(img, {19, 22}, {23, 30}, {26, 28}, bone);      // шипы
45     tri(img, {45, 22}, {41, 30}, {38, 28}, bone);
46     fillEllipse(img, {32, 20}, 7, 8, skin);             // голова
47     tri(img, {26, 15}, {28, 7}, {30, 14}, bone);       // рога
48     tri(img, {34, 14}, {36, 7}, {38, 15}, bone);
49     rect(img, 27, 19, 29, 21, rgb(255, 200, 30));      // глаза
50     rect(img, 35, 19, 37, 21, rgb(255, 200, 30));
51     rect(img, 28, 24, 36, 26, rgb(40, 10, 5));         // пасть
52     outline(img, rgb(25, 12, 6));
53     return img;
54 }
55
56 Image makeZombie(Pose pose) {
57     Image img(64, 64);
58     const auto cloth = rgb(75, 105, 55);
59     const auto gun = rgb(55, 55, 60);
60     const double s = stepOf(pose);
61     rect(img, 25 + s, 46, 30 + s, 63, rgb(60, 40, 25)); // сапоги

```

Листинг 21.4 (продолжение) — Монстры из эллипсов и многоугольников

```

monsters.cpp (продолжение) C++
62     rect(img, 34 - s, 46, 39 - s, 63, rgb(60, 40, 25));
63     fillEllipse(img, {32, 37}, 10, 12, cloth);           // туловище
64     rect(img, 22, 44, 42, 46, rgb(50, 40, 30));         // ремень
65     fillEllipse(img, {22, 37}, 3.5, 8, cloth);           // руки
66     fillEllipse(img, {42, 37}, 3.5, 8, cloth);
67     const std::array<Point, 4> rifle{{{19, 42}, {45, 30}, {46, 34},
68                                         {20, 46}}};
69     fillPolygon(img, rifle, gun);                         // винтовка
70     fillEllipse(img, {32, 20}, 6.5, 7.5, rgb(205, 155, 120)); // лицо
71     rect(img, 26, 13, 38, 15, rgb(50, 45, 40));         // стрижка
72     rect(img, 28, 19, 30, 21, rgb(200, 20, 20));        // глаза
73     rect(img, 34, 19, 36, 21, rgb(200, 20, 20));
74     outline(img, rgb(15, 20, 10));
75     return img;
76 }
77
78 const Image& monsterImage(MonsterKind kind, Pose pose) {
79     // Четыре кадра: [вид][поза]. Лямбда строит их при первом вызове.
80     static const auto frames = [] {
81         return std::array<std::array<Image, 2>, 2>{{{
82             {makeImp(Pose::Walk1), makeImp(Pose::Walk2)},
83             {makeZombie(Pose::Walk1), makeZombie(Pose::Walk2)},
84         }}};
85     }();
86     const auto k = static_cast<std::size_t>(kind);
87     const auto p = static_cast<std::size_t>(pose);
88     return frames[k][p];
89 }
90
91 } // namespace doom

```

Листинг 21.4 (продолжение) — Монстры из эллипсов и многоугольников

Строки 13–24 — две мелкие кисти поверх `fillPolygon`: прямоугольник и треугольник. Обрати внимание на `std::array<Point, 4> p{{{x0, y0}, ...}}}`: тройные фигурные скобки — внешние для массива, следующие для его внутреннего «сырого» массива, внутренние — для каждой точки. Две пары можно иногда опустить, но с тремя компилятор точно не перепутает.

Строки 26–29 — анимация. В двух позах ноги сдвинуты на 3 пикселя в разные стороны, а руки — на полтора. Чередует позы — и монстр переминается с ноги на ногу. В Doom у каждого монстра было по четыре кадра ходьбы, и смотрелось это ненамного богаче.

Строки 78–89 — все четыре картинки строятся один раз, при первом вызове. Здесь новый для нас приём: лямбда, которую тут же вызывают, — `[] { ... }()`. Она строит массив кадров и возвращает его, а `static` инициализируется результатом. Удобно, когда для инициализации нужно несколько строк кода, а не одно выражение.

Конец ознакомительного фрагмента.

Текст предоставлен ООО «Литрес».

Прочитайте эту книгу целиком, [купив полную легальную версию](#) на Литрес.

Безопасно оплатить книгу можно банковской картой Visa, MasterCard, Maestro, со счета мобильного телефона, с платежного терминала, в салоне МТС или Связной, через PayPal, WebMoney, Яндекс.Деньги, QIWI Кошелек, бонусными картами или другим удобным Вам способом.