

MINI DOOM

Современный C++ через игру



ТОМ 5

Шум, свет и
финальные титры

C++20 · Visual Studio 2026

18+

8+

Алексей Примак

**MINI DOOM современный
C++ через игру том 5 шум
свет и финальные титры**

«Автор»

2026

Примак А.

MINI DOOM современный C++ через игру том 5 шум свет и
финальные титры / А. Примак — «Автор», 2026

Заключительный том серии «MINI DOOM. Современный C++ через игру» превращает прототип в игру, которую не стыдно отдать другу. Звук синтезируется из синусов и шума и играет во втором потоке — с мьютексами, ThreadSanitizer и гонкой данных, пойманной вживую. Затем статус-бар с лицом героя, титульник с огнём из PSX-Doom, пауза, итоги уровня и автокарта. Уровни переезжают в текстовые файлы, исключения сообщают об ошибках, а поиск в ширину проверяет, проходим ли уровень. В финале — как работают взрослые: CMake, автотесты, робот-хаос, clang-tidy, Debug против Release. Плюс послесловие о настоящем Doom и список, куда расти дальше. Исходники всей серии — на GitHub. 18+.

Содержание

Вместо предисловия	5
Серия целиком	6
Часть V. Шум, свет и финальные титры	7
Глава 26. Звук из ничего	8
Звук — это числа	8
Четыре кирпичика	8
Огибающая и фильтр	10
Микшер	15
Второй поток	19
waveOut: звук в колонки	22
Итоги главы	25
Упражнения	26
Задачи	27
Глава 27. HUD и экраны	28
Конец ознакомительного фрагмента.	29

Алексей Примаков

MINI DOOM современный C++ через игру том 5 шум свет и финальные титры

Вместо предисловия

Это пятый том книги «MINI DOOM. Современный C++ через игру». К концу четвёртого тома в лабиринте живут монстры: спрайты с Z-буфером, имп и зомби, нарисованные кодом, пистолет и дробовик, конечный автомат, поиск в ширину, огненные шары и смерть с рестартом.

Остался последний слой: звук, экраны, уровни в файлах и то, как всё это собирают и проверяют взрослые. 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 V. Шум, свет и финальные титры

Четыре части позади, и у нас уже есть игра: лабиринт, двери, монстры, оружие, смерть. Но играть в неё — как смотреть фильм ужасов с выключенным звуком: вроде страшно, а вроде и нет. И начинается она сразу с выстрелов, без титульника, и кончается ничем — выхода с уровня нет.

Последняя часть превращает прототип в игру, которую не стыдно отдать другу. Звук — синтезированный кодом, как текстуры и монстры. Экраны: титульник с огнём, пауза, итоги уровня. Уровни в файлах, с проверкой, что их вообще можно пройти. И наконец, то, как всё это собирают и проверяют взрослые: SMake, автотесты, санитайзеры, статический анализ.

Глава 26. Звук из ничего

Попробуй вспомнить Doom без звука. Не получится: рычание импа из темноты, сухой хлопок дробовика, лязг двери — это половина атмосферы. Хорошая новость: для звука, как и для текстур, файлы не нужны. Любой звук — это просто последовательность чисел, а числа мы умеем считать.

В этой главе ты узнаешь, что такое сэмпл и из каких «кирпичиков» складываются звуки, напишешь синтезатор с огибающей и фильтром, микшер, который играет несколько звуков сразу, — и впервые запустишь в своей программе **второй поток**. А вместе с ним встретишь самую подлую ошибку многопоточности — гонку данных.

Звук — это числа

Звук — это колебания давления воздуха. Динамик двигает диффузор вперёд-назад, тот толкает воздух, воздух — барабанную перепонку. Компьютер управляет динамиком очень просто: много раз в секунду говорит ему «встань в такое-то положение». Каждое такое число называется **сэмплом** (sample — отсчёт), а сколько их в секунду — **частотой дискретизации**.

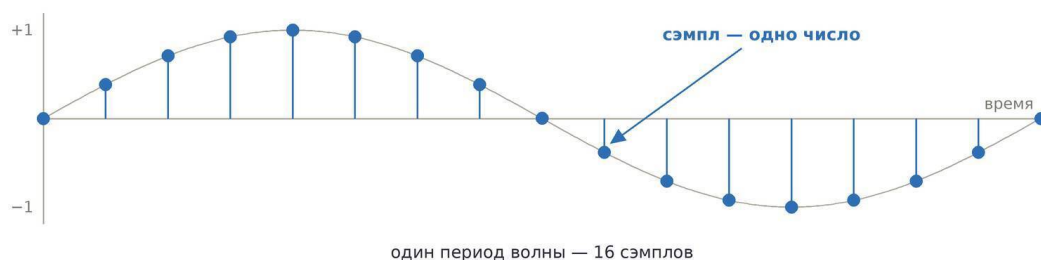


Рисунок 26.1 — Волна и её сэмплы: компьютер помнит только синие точки

Наш синтезатор работает на частоте 22 050 сэмплов в секунду — вдвое чаще, чем звук в Doom (11 025), и вдвое реже, чем компакт-диск (44 100). Сэмпл внутри программы — float от -1 до 1 , а на звуковую карту уходит 16-битное целое от $-32\,768$ до $32\,767$. По рисунку 26.1 видно главное: чем больше сэмплов на период, тем точнее волна. Частота дискретизации должна быть хотя бы вдвое выше самого высокого звука, который хочешь передать, — так что 22 050 честно передают звуки до 11 кГц. Для взрывов и рычания хватает с запасом.

Четыре кирпича

Любой синтезатор, от игрушечного до студийного, начинается с **генераторов** — функций, которые рисуют одну волну. У волны есть **фаза** — где мы сейчас внутри периода, число от 0 до 1. Генератор по фазе отвечает, где диффузор. Классических форм четыре, и три из них — в листинге 26.1.

```

main.cpp C++
1  #include <cmath>
2  #include <format>
3  #include <iostream>
4  #include <numbers>
5
6  // Четыре формы волны: значение в точке phase от 0 до 1.
7  double sine(double phase) {
8      return std::sin(2.0 * std::numbers::pi * phase);
9  }
10 double saw(double phase) { return 2.0 * phase - 1.0; }
11 double square(double phase) { return phase < 0.5 ? 1.0 : -1.0; }
12
13 int main() {
14     std::cout << "phase    sine    saw    square\n";
15     for (int i = 0; i < 8; ++i) {
16         const double p = i / 8.0;
17         std::cout << std::format("{:5.3f} {:6.2f} {:6.2f} {:7.2f}\n",
18                                 p, sine(p) + 0.0, saw(p),
19                                 square(p));
20     }
21     // Сэмпл — это значение волны в момент времени. 440 Гц, ля:
22     constexpr int kRate = 22050;
23     std::cout << "A4 at 22050 Hz:";
24     for (int i = 0; i < 6; ++i) {
25         const double phase = 440.0 * i / kRate; // сколько периодов
26         const double s = sine(phase - std::floor(phase));
27         std::cout << ' ' << static_cast<int>(s * 32767);
28     }
29     std::cout << '\n';
30 }

```

```

$ ./a.out
phase    sine    saw    square
0.000    0.00   -1.00    1.00
0.125    0.71   -0.75    1.00
0.250    1.00   -0.50    1.00
0.375    0.71   -0.25    1.00
0.500    0.00    0.00   -1.00
0.625   -0.71    0.25   -1.00
0.750   -1.00    0.50   -1.00
0.875   -0.71    0.75   -1.00
A4 at 22050 Hz: 0 4097 8130 12036 15752 19222

```

Листинг 26.1 — Синус, пила, меандр — и первые сэмплы ноты ля

Синус — чистый, мягкий тон, как свист. **Пила** растёт от -1 до 1 и падает обратно — звук резкий, «жужжащий», богатый обертонами: из неё получается рычание. **Меандр** (квадратная волна) прыгает между $+1$ и -1 — это голос старых приставок, «8-бит». Последняя строка

вывода — шесть первых сэмплов ноты ля (440 Гц) в 16-битном виде: за шесть отсчётов волна прошла едва восьмую часть периода, ведь период — это $22\,050 / 440 \approx 50$ сэмплов.

Четвёртый кирпичик — **шум**: случайное число в каждом сэмпле. Чистый шум — это шипение радио между станциями. Но если менять случайное значение не каждый сэмпл, а, скажем, 800 раз в секунду, а между сменами держать прежнее, шум получает «высоту»: низкий хриплый гул вместо шипения. Это приём «sample and hold», и из него получаются выстрелы и взрывы. Все четыре формы — на рисунке 26.2.

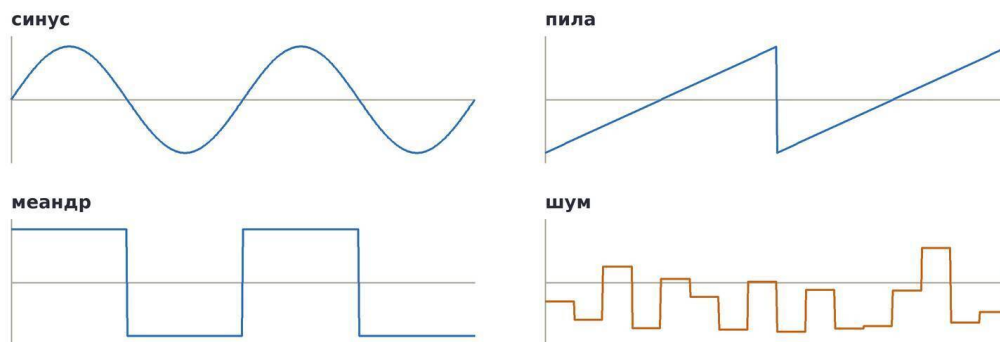


Рисунок 26.2 — Два периода каждой формы. Шум — ступеньки случайной высоты

Огибающая и фильтр

Генератор звучит бесконечно и одинаково громко. Настоящий звук так не делает: выстрел вспыхивает мгновенно и затихает, рычание нарастает и тянется. Форму громкости во времени задаёт **огибающая** — ещё одна функция, на которую умножаем каждый сэмпл. Наша — простейшая: короткий линейный подъём («атака») и плавное затухание по параболе, рисунок 26.3.



Рисунок 26.3 — Огибающая: громкость звука от начала до конца

Второй инструмент — **фильтр нижних частот**. Он срезает высокие, «пискливые» частоты и оставляет низкие. Устроен он изумительно просто — листинг 26.2, строки 57–68: выход у на каждом шаге догоняет вход на долю a . Представь тяжёлую лодку на верёвке: дёргай верёвку часто-часто — лодка почти не сдвинется (высокие частоты пропали), тяни медленно — лодка послушно идёт следом (низкие прошли). Чем ниже частота среза cutoff, тем меньше a и тем «тяжелее» лодка: звук становится глухим, как из-за стены.

```

synth.cpp
47 // Громкость во времени: быстро вверх, потом плавно в ноль.
48 double envelope(double t, double attack, double length) {
49     if (t < attack) {
50         return t / attack;
51     }
52     const double u = (t - attack) / (length - attack);
53     return (1.0 - u) * (1.0 - u);
54 }
55
56 // Фильтр нижних частот: срезает «писк», оставляет «бас».
57 void lowpass(Wave& w, double cutoff) {
58     if (cutoff <= 0.0) {
59         return;
60     }
61     const double w0 = 2.0 * std::numbers::pi * cutoff / kSampleRate;
62     const double a = 1.0 - std::exp(-w0);
63     double y = 0.0;
64     for (float& s : w) {
65         y += a * (s - y); // догоняем вход на долю a
66         s = static_cast<float>(y);
67     }
68 }

```

Листинг 26.2 — Огибающая и фильтр нижних частот

Остаётся собрать всё вместе. Звук в нашей игре — **рецепт**: два слоя (генератор, начальная и конечная частота, громкость), атака, длина и частота среза фильтра, листинг 26.3. Частота слоя за время звука плавно едет от from к to: выстрел начинается высоким треском и проседает в бас, рык импа сползает вниз на октаву.

```

synth.hpp
C++
15  enum class Shape { Sine, Saw, Square, Noise };
16
17  // Один голос: форма волны и частота, которая плавно едет
18  // от from к to за время звука.
19  struct Layer {
20      Shape shape = Shape::Sine;
21      double from = 440.0;          // Гц в начале
22      double to = 440.0;           // Гц в конце
23      double gain = 0.0;           // 0 — слоя нет
24  };
25
26  struct Recipe {
27      std::array<Layer, 2> layers;
28      double attack = 0.01;        // с: нарастание громкости
29      double length = 0.5;         // с: весь звук
30      double cutoff = 0.0;         // Гц: фильтр; 0 — без фильтра
31  };

```

Листинг 26.3 — Слой и рецепт звука

Двенадцать рецептов — таблица прямо в коде, constexpr-массив в порядке перечисления SoundId. Вот первые четыре, листинг 26.4. Писать их — чистое удовольствие: меняешь число, собираешь, слушаешь.

```

synth.cpp
C++
11  namespace {
12      using enum Shape;
13
14      // Рецепты в порядке SoundId: два слоя, атака, длина, фильтр.
15      constexpr std::array<Recipe, kSoundCount> kRecipes{{
16          {{{Noise, 8000, 2000, 1.0}, {Square, 180, 60, 0.5}}},
17          {0.002, 0.25, 5000}, // Pistol
18          {{{Noise, 6000, 800, 1.0}, {Saw, 90, 40, 0.7}}},
19          {0.002, 0.60, 3000}, // Shotgun
20          {{{Saw, 220, 110, 0.7}, {Noise, 1500, 600, 0.4}}},
21          {0.05, 0.70, 1800}, // ImpSight
22          {{{Square, 110, 80, 0.6}, {Noise, 900, 500, 0.4}}},
23          {0.05, 0.60, 1200}, // ZombieSight

```

Листинг 26.4 — Рецепты: пистолет, дробовик, имп, зомби

Строка 12 — using enum Shape из C++20: внутри этой области можно писать Noise вместо Shape::Noise. В таблице это экономит половину ширины строки.

Сам синтез — листинг 26.5. Для каждого слоя идём по сэмплам, двигаем фазу на $\text{freq} / \text{kSampleRate}$ (столько периодов проходит за один сэмпл) и складываем значение генератора в общий буфер. Потом фильтр, огибающая и нормировка: все звуки приводим к пику 0,8, чтобы ни один не был тише остальных просто потому, что так получилось.

```

synth.cpp
C++

90 Wave synthesize(const Recipe& r, unsigned seed) {
91     const auto n = static_cast<std::size_t>(r.length * kSampleRate);
92     Wave out(n, 0.0f);
93     std::minstd_rand rng(seed);
94     std::uniform_real_distribution<float> white(-1.0f, 1.0f);
95     for (const Layer& layer : r.layers) {
96         if (layer.gain == 0.0) {
97             continue;
98         }
99         double phase = 0.0;
100        float held = 0.0f; // шум меняется раз в период
101        for (std::size_t i = 0; i < n; ++i) {
102            const double u =
103                static_cast<double>(i) / static_cast<double>(n);
104            // Частота едет по степени: так её слышит ухо.
105            const double freq =
106                layer.from * std::pow(layer.to / layer.from, u);
107            phase += freq / kSampleRate;
108            if (phase >= 1.0) {
109                phase -= 1.0;
110                held = white(rng);

```

Листинг 26.5 — Синтез звука по рецепту

```

synth.cpp (продолжение)
C++

111     }
112     out[i] += static_cast<float>(layer.gain) *
113         oscillator(layer.shape, phase, held);
114 }
115 }
116 lowpass(out, r.cutoff);
117 float peak = 0.0f;
118 for (std::size_t i = 0; i < n; ++i) {
119     const double t = static_cast<double>(i) / kSampleRate;
120     const double env = envelope(t, r.attack, r.length);
121     out[i] *= static_cast<float>(env);
122     peak = std::max(peak, std::abs(out[i]));
123 }
124 if (peak > 0.0f) { // все звуки одной громкости
125     for (float& s : out) {
126         s *= 0.8f / peak;
127     }
128 }
129 return out;
130 }

```

Листинг 26.5 (продолжение) — Синтез звука по рецепту

Строки 104–106. Частота едет не по прямой, а по степени: $\text{from} \cdot (\text{to} / \text{from})^u$. Ухо слышит не герцы, а их отношения: $110 \rightarrow 220$ Гц и $220 \rightarrow 440$ Гц — это одинаковый шаг, октава. При линейном спуске звук сначала почти стоял бы на месте, а в конце рушился вниз.

Строки 107–111. Когда фаза перевалила за 1, период закончился: вычитаем единицу и, для шума, берём новое случайное число. Вот и весь sample and hold.

Строка 93. Генератор случайных чисел у каждого звука свой, с собственным зерном, — шум каждый раз одинаковый, и звуки не меняются от запуска к запуску. Для тестов это важно так же, как для монстров в главе 22.

Послушаем, что получилось. Тест умеет записывать все звуки в WAV-файлы — листинг 26.6. Файлы лежат и в архиве примеров, в папке ch26/wav: открой любой плеером.

```

терминал SHELL
$ g++ -std=c++20 -O2 ai.cpp art.cpp canvas.cpp game.cpp hud.cpp map.cpp
  mixer.cpp monsters.cpp noise.cpp paint.cpp pathfind.cpp raycast.cpp
  sprites.cpp synth.cpp test.cpp text.cpp textures.cpp topview.cpp
  view3d.cpp weapons.cpp -o sounds
$ ./sounds wav
      pistol.wav    0.25 s
      shotgun.wav   0.60 s
      imp_sight.wav  0.70 s
      zombie_sight.wav 0.60 s
      monster_pain.wav 0.30 s
      monster_death.wav 1.00 s
      fireball.wav   0.50 s
      boom.wav       0.90 s
      door.wav       0.90 s
      pickup.wav     0.15 s
      player_hurt.wav 0.35 s
      player_death.wav 1.40 s

```

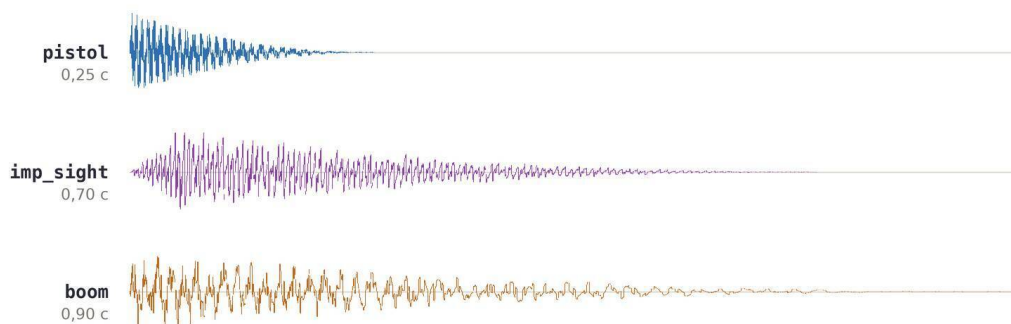
Листинг 26.6 — Синтезируем все звуки игры и записываем в WAV

Рисунок 26.4 — Настоящие звуки из WAV-файлов: выстрел, рык импа, взрыв

На рисунке 26.4 видна огибающая каждого звука: выстрел — короткий удар, рык импа нарастает и тянется, взрыв бьёт резко и долго гаснет. А внутри видны и слои: у взрыва в начале частый шум, к концу — медленные волны низкого синуса.

Лирика. Звуки Doom — 8-битные, 11 025 сэмплов в секунду. Многие из них не записаны специально для игры, а взяты из коммерческих библиотек звуковых эффектов и переделаны: замедлены, склеены, пропущены через эффекты. Поэтому внимательные игроки до сих пор узнают «голоса» импов и дверей Doom в фильмах и других играх девяностых.

Микшер

В бою звучит много всего сразу: твой дробовик, рык импа, взрыв шара, лязг двери. А динамик один, и в каждый момент он может стоять только в одном положении. Решение — **сложить**: колебания воздуха от разных источников в жизни тоже просто складываются. Программа, которая это делает, называется **микшер**, листинг 26.7.

```

8 void Mixer::play(SoundId id, float volume) {
9     const std::scoped_lock lock(mutex_);
10    if (voices_.size() == kMaxVoices) {
11        voices_.erase(voices_.begin()); // самый старый — вон
12    }
13    voices_.push_back({&bank_[id], 0, volume});
14 }
15
16 void Mixer::mix(std::span<std::int16_t> out) {
17     const std::scoped_lock lock(mutex_);
18     for (std::size_t i = 0; i < out.size(); ++i) {
19         float sum = 0.0f;
20         for (Voice& v : voices_) {
21             if (v.pos < v.wave->size()) {
22                 sum += (*v.wave)[v.pos++] * v.volume;
23             }
24         }
25         // Два громких звука в сумме вылезут за 1 — обрезаем.
26         sum = std::clamp(sum, -1.0f, 1.0f);
27         out[i] = static_cast<std::int16_t>(sum * 32767.0f);
28     }
29     std::erase_if(voices_, [](const Voice& v) {
30         return v.pos >= v.wave->size(); // доиграл — свободен
31     });
32 }

```

Листинг 26.7 — Микшер: сложить все звучащие голоса

Каждый звучащий звук — **голос** (Voice): указатель на волну из банка звуков, номер следующего сэмпла и громкость. `play` добавляет голос, `mix` заполняет буфер для звуковой карты: для каждого сэмпла складывает текущие сэмплы всех голосов, умноженные на громкость, и двигает их вперёд. Доигравшие голоса удаляем — знакомым `std::erase_if`. Голосов не больше 16: если места нет, самый старый замолкает. Doom поступал так же — у него было всего 8 каналов.

Строка 26 — важная. Два звука с пиком 0,8 в сумме дают 1,6, а 16-битное целое не вмещает больше 32 767. Что будет без `std::clamp` — узнаешь в упражнении 26.2, и это стоит увидеть. С `clamp` громкие места просто срезаются: звук слегка «хрипит», но не превращается в мусор.

Остаётся вопрос: откуда микшер узнаёт, что пора играть? Game про звук ничего не знает и знать не должна — это та же стена между игрой и платформой, что и в главе 12. Поэтому игра лишь записывает **события**: «прозвучал выстрел, громкость 0,75» — листинг 26.8.

```

game.cpp
198 void Game::openDoor(Door& door, int cx, int cy) {
199     if (door.move <= 0 && door.open < 1.0) { // ещё не ехала
200         sound(SoundId::Door, cx + 0.5, cy + 0.5);
201     }
202     door.move = 1;
203 }
204
205 void Game::sound(SoundId id, double x, double y) {
206     const double dist = std::hypot(x - x_, y - y_);
207     const double volume = 1.0 - dist / kEarshot; // линейно в ноль
208     if (volume > 0.0) {
209         sounds_.push_back({id, static_cast<float>(volume)});
210     }
211 }

```

Листинг 26.8 — Звук в точке мира: чем дальше, тем тише

Громкость падает с расстоянием линейно и обнуляется на 16 клетках. Звуки дверей, выстрелов, взрывов, рыка — всё идёт через `sound`. `openDoor` заодно следит, чтобы дверь лязгала один раз, когда начинает открываться, а не каждый кадр, пока её держат открытой.

Главный цикл раз в кадр забирает накопленные события методом `takeSounds`:

```

game.hpp
52 // Звуки, прозвучавшие с прошлого вызова; очередь очищается.
53 [[nodiscard]] std::vector<SoundEvent> takeSounds() {
54     return std::exchange(sounds_, {});
55 }

```

Листинг 26.9 — Забрать события и очистить очередь

`std::exchange(sounds_, {})` делает две вещи одним махом: кладёт в `sounds_` пустой вектор и возвращает то, что там лежало раньше. Без копирования — вектор перемещается. Это стандартная идиома «забрать и обнулить».

Автотест, листинг 26.10, повторяет бой из главы 25 — игрок стоит в коридоре под огнём импов — и выписывает все события за две секунды. Звук здесь не нужен: проверяем логику.

```

test.cpp C++
55     for (std::size_t i = 0; i < doom::kSoundCount; ++i) {
56         const auto id = static_cast<doom::SoundId>(i);
57         render(bank, id);
58         const auto n = static_cast<double>(bank[id].size());
59         const double sec = n / doom::kSampleRate;
60         std::cout << std::format("{:>13}.wav  {:5.2f} s\n",
61                                 doom::soundName(id), sec);
62     }
63     return 0;
64 }
65 // Бой из главы 25: стоим под огнём и выписываем все звуки.
66 doom::Game game{doom::Map{doom::kLevel}};
67 game.teleport(3.5, 11.5, 0.0);
68 doom::Mixer mixer(bank);
69 for (int frame = 1; frame <= 120; ++frame) {
70     game.update({.fire = frame == 30}, 1.0 / 60.0);
71     for (const doom::SoundEvent& e : game.takeSounds()) {
72         mixer.play(e.id, e.volume);
73         std::cout << std::format(
74             "t {:4.2f}  {:<13} vol {:.2f}\n", frame / 60.0,
75             doom::soundName(e.id), e.volume);
76     }
77     std::vector<std::int16_t> chunk(doom::kSampleRate / 60);
78     mixer.mix(chunk); // кадр звука
79 }
80 std::cout << std::format("still playing: {}\n", mixer.playing());
81 }

```

```

$ ./a.out
t 0.02  imp_sight      vol 0.75
t 0.02  zombie_sight   vol 0.38
t 0.02  imp_sight      vol 0.80
t 0.33  fireball       vol 0.75
t 0.33  fireball       vol 0.80
t 0.50  pistol         vol 1.00
t 0.50  monster_pain   vol 0.75
t 0.73  player_hurt    vol 1.00
t 0.73  boom           vol 0.98
t 0.87  player_hurt    vol 1.00
t 0.87  boom           vol 0.98
t 1.07  fireball       vol 0.75
t 1.60  player_hurt    vol 1.00
t 1.60  boom           vol 0.98
still playing: 1

```

Листинг 26.10 — Какие звуки рождает бой

— **t 0.02** — два импа и зомби заметили игрока. Громкость рыка — по расстоянию: ближний имп 0,80, дальний 0,75, зомби в другом конце зала — 0,38.

- **t 0.33** — оба импа бросили шары, слышно, как они вспыхнули.
- **t 0.50** — наш выстрел (кадр 30) — 1,00, это мы сами. Следом — `monster_pain`: попали в импа.
- **t 0.73 и 0.87** — шары долетели: вскрик игрока и взрыв почти на полной громкости — он ведь рядом.
- **still playing: 1** — в конце двух секунд ещё не доиграл последний звук.

Второй поток

Звуковой карте нужны данные постоянно: 22 050 сэмплов каждую секунду, без пауз. Опоздаешь на пару миллисекунд — в колонках щелчок или заикание. А игровой цикл живёт своей жизнью: кадр может затянуться, окно — свернуться, Windows — задуматься. Звук нельзя кормить из игрового цикла, ему нужен собственный исполнитель, который занимается только им. Это **поток** (thread).

До сих пор у каждой нашей программы был ровно один поток: процессор шёл по коду строчка за строчкой. Второй поток — это второй «исполнитель», который идёт по своей функции одновременно с первым, на другом ядре процессора. В C++20 его создают одной строкой: `std::jthread t(функция)`. Буква `j` — от `join`: когда объект `jthread` уничтожается, он дожидается, пока поток закончит работу. Старый `std::thread` в такой ситуации просто роняет программу.

Звучит прекрасно, но у двух исполнителей появляется общая проблема — общие данные. Вот простейший опыт, листинг 26.11: два потока по миллиону раз прибавляют единицу к одной переменной.

```

1  #include <iostream>
2  #include <thread>
3
4  int main() {
5      int counter = 0;
6      // Два потока по миллиону раз прибавляют единицу к одной
7      // переменной. Сколько получится?
8      {
9          std::jthread a([&counter] {
10             for (int i = 0; i < 1'000'000; ++i) {
11                 ++counter;
12             }
13         });
14         std::jthread b([&counter] {
15             for (int i = 0; i < 1'000'000; ++i) {
16                 ++counter;
17             }
18         });
19     } // здесь jthread сами ждут завершения потоков
20     std::cout << "counter = " << counter << '\n';
21 }

```

```

$ ./a.out
counter = 1795418

```

Листинг 26.11 — Два потока, одна переменная

Должно быть 2 000 000. Получилось почти вдвое меньше, и при каждом запуске — другое число. Дело в том, что `++counter` — это не одно действие, а три: прочитать значение из памяти, прибавить единицу, записать обратно. Если два потока прочитали одно и то же значение, оба запишут одно и то же, и одна прибавка пропадёт, рисунок 26.5.

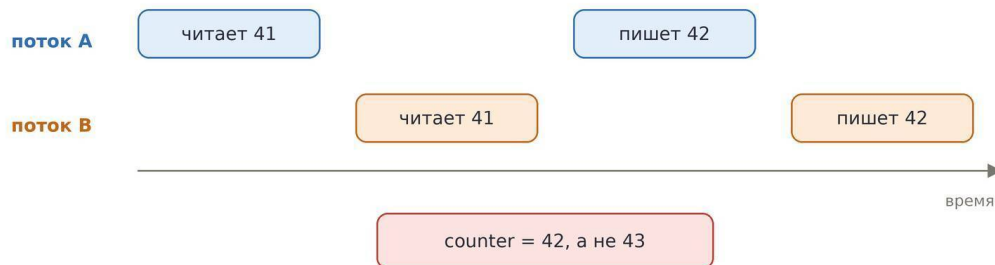


Рисунок 26.5 — Гонка данных: два потока прибавили по единице, а счётчик вырос на одну

Это называется **гонка данных** (data race): два потока одновременно обращаются к одной памяти, и хотя бы один пишет. По стандарту C++ гонка данных — неопределённое поведение, как выход за границу массива. Счётчик, который «немного недосчитал», — лучший из исходов. Хуже, когда один поток читает вектор, пока второй его расширяет: тогда первый читает освобождённую память.

Самое противное — гонки не воспроизводятся. Программа может месяц работать и упасть у одного игрока из тысячи. Поэтому для них, как и для памяти, есть санитайзер: **ThreadSanitizer**. Он следит за каждым обращением к памяти из каждого потока, листинг 26.12.

```
терминал SHELL
# собираем с санитайзером потоков
$ g++ -std=c++20 -g -fsanitize=thread main.cpp
$ ./a.out
WARNING: ThreadSanitizer: data race
Read of size 4 by thread T1:
    #0 operator() main.cpp:11
Previous write of size 4 by thread T2:
    #0 operator() main.cpp:16
SUMMARY: ThreadSanitizer: data race main.cpp:11 in operator()
counter = 2000000
ThreadSanitizer: reported 2 warnings
# ... отчёт сокращён до сути
```

Листинг 26.12 — ThreadSanitizer ловит гонку

Под санитайзером счёт вдруг сошёлся — он сильно замедляет программу, и потоки реже сталкиваются. Но гонку он всё равно увидел: это его главная сила, ему не нужно, чтобы ошибка проявилась. Санитайзер называет обе строки: поток T2 читает counter в строке 16, а поток T1

перед этим писал в строке 11, и между ними нет никакой синхронизации. Ровно то, что на рисунке 26.5.

Лечение — **мьютекс** (mutex, от mutual exclusion — взаимное исключение). Это замок с одним ключом: кто взял ключ, тот работает с данными, остальные ждут у двери. В C++ ключ берут объектом `std::scoped_lock`: конструктор запирает мьютекс, деструктор отпирает. Это RAII из главы 7 — забыть отпереть невозможно, даже если внутри вылетит исключение. Листинг 26.13.

```

1  #include <iostream>
2  #include <mutex>
3  #include <thread>
4
5  int main() {
6      int counter = 0;
7      std::mutex m;                // замок на counter
8      const auto work = [&] {
9          for (int i = 0; i < 1'000'000; ++i) {
10             const std::scoped_lock lock(m); // взяли ключ...
11             ++counter;
12         }                                // ...и отдали
13     };
14     {
15         std::jthread a(work);
16         std::jthread b(work);
17     }
18     std::cout << "counter = " << counter << '\n';
19 }

```

```

$ ./a.out
counter = 20000000

```

Листинг 26.13 — Тот же счётчик под замком

Запомни. Если данные трогают два потока и хотя бы один из них пишет — каждое обращение должно быть под одним и тем же мьютексом. Даже чтение. Особенно чтение: «я же только читаю» — классические последние слова перед гонкой.

Теперь понятно, зачем в микшере `mutex_` и почему его берут и `play`, и `mix`, и даже `playing`, которая только читает размер вектора. `play` вызывает поток игры, `mix` — звуковой поток, и вектор голосов у них общий. Вся схема — на рисунке 26.6.



Рисунок 26.6 — Два потока и микшер между ними

Обрати внимание, чего игра **не** делает: не вызывает микшер из глубины update, посреди расчётов. Она копит события у себя, а главный цикл отдаёт их пачкой. Чем реже и короче потоки встречаются у общего замка, тем меньше они друг друга ждут.

waveOut: звук в колонки

Последний шаг — отдать сэмплы Windows. Способов несколько, и самый старый и простой — **waveOut**, ровесник Windows 3.1. Он работает так: мы даём ему буфер с сэмплами, он его проигрывает и ставит в заголовке буфера флаг WHDR_DONE — «доиграл». Чтобы не было пауз, буферов три и они ходят по кругу, рисунок 26.7: один звучит, второй ждёт своей очереди, третий мы заполняем.

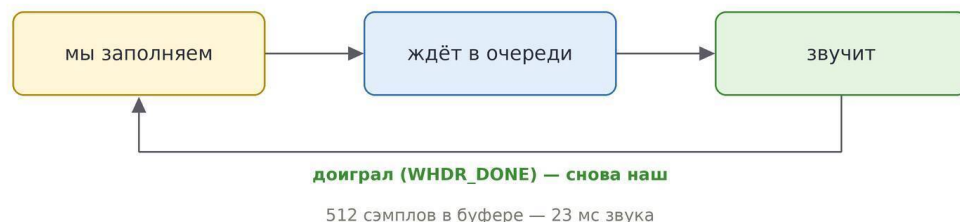


Рисунок 26.7 — Три буфера по кругу: звуковая карта не ждёт ни миллисекунды

Код — листинг 26.14. Это единственный файл главы, который знает про Windows; в автотест под Linux он не входит, как и main_win32.cpp.

```
audio_win32.cpp C++
14 AudioOut::AudioOut(Mixer& mixer)
15     : mixer_(mixer),
16       thread_([this](std::stop_token stop) { run(stop); }) {}
17
18 void AudioOut::run(std::stop_token stop) {
19     WAVEFORMATEX fmt{};
20     fmt.wFormatTag = WAVE_FORMAT_PCM;
21     fmt.nChannels = 1; // моно
22     fmt.nSamplesPerSec = kSampleRate;
23     fmt.wBitsPerSample = 16;
24     fmt.nBlockAlign = 2; // байт на сэмпл
25     fmt.nAvgBytesPerSec = kSampleRate * 2;
26     const HANDLE done = CreateEventW(nullptr, FALSE, FALSE, nullptr);
27     HWAVEOUT out = nullptr;
28     if (waveOutOpen(&out, WAVE_MAPPER, &fmt,
29                    reinterpret_cast<DWORD_PTR>(done), 0,
30                    CALLBACK_EVENT) != MMSYSERR_NOERROR) {
31         CloseHandle(done);
32         return; // нет звука — играем молча
33     }
34     // Три буфера по 512 сэмплов (23 мс): один звучит,
35     // второй ждёт очереди, третий мы заполняем.
36     constexpr std::size_t kChunk = 512;
37     std::array<std::array<std::int16_t, kChunk>, 3> data{};
```

Листинг 26.14 — Звуковой поток: waveOut и три буфера

```

audio_win32.cpp (продолжение)
38     std::array<WAVEHDR, 3> hdr{};
39     for (std::size_t i = 0; i < hdr.size(); ++i) {
40         hdr[i].lpData = reinterpret_cast<LPSTR>(data[i].data());
41         hdr[i].dwBufferLength = static_cast<DWORD>(sizeof(data[i]));
42         waveOutPrepareHeader(out, &hdr[i], sizeof(WAVEHDR));
43         hdr[i].dwFlags |= WHDR_DONE;    // «свободен» — заполнить
44     }
45     while (!stop.stop_requested()) {
46         for (std::size_t i = 0; i < hdr.size(); ++i) {
47             if ((hdr[i].dwFlags & WHDR_DONE) != 0) {
48                 mixer_.mix(data[i]);
49                 waveOutWrite(out, &hdr[i], sizeof(WAVEHDR));
50             }
51         }
52         WaitForSingleObject(done, 20); // доиграл буфер — разбудят
53     }
54     waveOutReset(out);                // стоп и вернуть буферы
55     for (WAVEHDR& h : hdr) {
56         waveOutUnprepareHeader(out, &h, sizeof(WAVEHDR));
57     }
58     waveOutClose(out);
59     CloseHandle(done);
60 }

```

Листинг 26.14 (продолжение) — Звуковой поток: waveOut и три буфера

Строки 14–16. Поток запускается прямо в списке инициализации конструктора: `thread_` получает лямбду, которая вызовет `run`. Параметр `std::stop_token` — подарок `jthread`: через него поток узнаёт, что его просят остановиться. Деструктор `jthread` сам вызывает `request_stop`, а потом ждёт, пока `run` вернётся. Никаких флагов «пора выходить» руками.

Строки 28–33. `waveOutOpen` открывает звуковое устройство и обещает дёргать событие `done` каждый раз, когда буфер доиграл. Если звуковой карты нет (так бывает — например, на сервере сборки), просто выходим: игра будет беззвучной, но не упадёт.

Строки 45–53 — сердце потока. Проверяем каждый буфер: доиграл — заполняем свежими сэмплами из микшера и отдаём обратно. Потом спим, пока Windows не разбудит событием, но не дольше 20 мс, чтобы не прозевать просьбу остановиться.

Строки 54–59. Перед выходом — всё вернуть: `waveOutReset` останавливает воспроизведение и отдаёт буферы, потом разбираем заголовки и закрываем устройство. Буферы живут на стеке `run`, и выйти, пока Windows ещё в них пишет, — верный способ получить порчу памяти.

Габри. В классе `AudioOut` поле `thread_` объявлено последним — и не случайно. Поля создаются сверху вниз, а уничтожаются снизу вверх. Поток стартует в конструкторе и сразу начинает пользоваться `mixer_`. Объяви `thread_` первым — поток может запуститься раньше, чем `mixer_` будет инициализирован. А при уничтожении первым умрёт уже `mixer_`, пока поток ещё работает. Правило: поток — последнее поле класса.

В `main_win32.cpp` добавилось шесть строк, листинг 26.15: банк звуков, микшер, вывод звука — и раздача событий после `update`.

```

main_win32.cpp
226     const doom::SoundBank bank;           // синтез всех звуков
227     doom::Mixer mixer(bank);
228     doom::AudioOut audio(mixer);          // отсюда звучит второй
      поток
229     doom::Game game{doom::Map{doom::kLevel}};

```

Листинг 26.15 — Звук в главном цикле

```

main_win32.cpp
257     game.update(collectInput(), dt);
258     for (const doom::SoundEvent& e : game.takeSounds()) {
259         mixer.play(e.id, e.volume);
260     }

```

Листинг 26.16 — Раздача событий микшеру

Порядок объектов в `main` тоже важен по той же причине: `audio` объявлен после `mixer`, а `mixer` — после `bank`. Уничтожатся они в обратном порядке: сначала остановится поток, потом исчезнет микшер, последним — банк звуков. Запусти игру, выстрели — и услышишь, как имп отзовется из-за стены.

По-взрослому. `waveOut` — дедушка звука в Windows. Современные игры пишут через WASAPI или XAudio2, а чаще — через готовые движки: FMOD, Wwise, бесплатный `miniaudio`. Они дают 3D-звук, эффекты эха, потоковую музыку и задержку в несколько миллисекунд. И ещё одно: в настоящем звуковом потоке мьютексов избегают — если поток игры задержится с замком в руках, звук заикнётся. Вместо замков используют очереди без блокировок (`lock-free`). Но начинать надо с мьютекса: он прост и честен, а `lock-free` код пишут, когда точно знают, зачем.

Итоги главы

— Звук — это сэмплы: 22 050 чисел в секунду от -1 до 1 , на карту — 16-битными целыми.

— Кирпичики синтеза: синус, пила, меандр, шум; огибающая задаёт громкость во времени, фильтр срезает высокие частоты.

— Звук игры — рецепт из двух слоёв; таблица рецептов — `constexpr`-массив, `using enum` сокращает имена.

— Микшер складывает голоса и обрезает сумму `std::clamp`. Игра только записывает события; `std::exchange` забирает их пачкой.

— Звук живёт во втором потоке: `std::jthread` сам ждёт завершения, `std::stop_token` сообщает об остановке.

— Общие данные двух потоков — только под мьютексом; `std::scoped_lock` запирает и отпирает сам. Гонки ловит ThreadSanitizer.

— `waveOut`: три буфера по кругу. Поле с потоком — последнее в классе.

Упражнения

Упражнение 26.1. *Что выведет?* Листинг 26.17.



```

1  #include <iostream>
2  #include <string>
3  #include <utility>
4  #include <vector>
5
6  int main() {
7      std::vector<std::string> sounds{"pistol", "boom"};
8      auto taken = std::exchange(sounds, {"door"});
9      std::cout << taken.size() << ' ' << sounds.size() << ' '
10               << sounds.front() << '\n';
11  }
    
```

Листинг 26.17 — *Что выведет программа?*

Упражнение 26.2. *Найди баг.* Микшер без `std::clamp` складывает два громких звука, листинг 26.18. Сумма 1,5 — значит, на выходе должно быть что-то около максимума, 32 767. Почему числа такие?

```

1  #include <cstdlib>
2  #include <iostream>
3  #include <vector>
4
5  // Микшер без защиты: складываем два громких звука.
6  int main() {
7      const std::vector<float> shot{0.8f, 0.9f, -0.7f};
8      const std::vector<float> boom{0.7f, 0.6f, -0.8f};
9      for (std::size_t i = 0; i < shot.size(); ++i) {
10         const float sum = shot[i] + boom[i];
11         const auto s = static_cast<std::int16_t>(sum * 32767.0f);
12         std::cout << s << ' ';
13     }
14     std::cout << '\n';
15 }

```

```

$ ./a.out
-16386 -16386 16386

```

Листинг 26.18 — Громко + громко = тихо и наоборот?

Упражнение 26.3. *Напиши сам.* Сделай звук стереофоническим: монстр слева должен звучать громче в левой колонке. Подсказка: угол на источник относительно взгляда — `atan2`, а `waveOut` умеет два канала.

Упражнение 26.4. *Подумай.* Метод `Mixer::playing` только читает размер вектора. Зачем ему мьютекс? Что может случиться без него?

Задачи

Задача 26.1. Мягкое ограничение. `std::clamp` срезает пики углом, и громкие места хрипят. Заменяй его на `std::tanh(sum)`: малые значения он почти не трогает, а большие плавно прижимает к ± 1 . Послушай разницу на двух одновременных взрывах.

Задача 26.2. Звук шагов. Придумай рецепт короткого глухого шага и проигрывай его, пока игрок идёт, примерно дважды в секунду. Подсказка: фаза покачивания оружия `bob_` из главы 22 уже считает шаги.

Задача 26.3. Не больше двух одинаковых. Когда очередь дробовика бьёт по толпе, звучит шесть одинаковых `monster_pain` — каша. Сделай, чтобы одного звука одновременно звучало не больше двух экземпляров: третий заменяет самый старый из них.

Глава 27. HUD и экраны

Наша игра до сих пор похожа на техно-демку: запустил — и ты уже в лабиринте, цифры здоровья висят по углам, а выйти с уровня некуда. Настоящая игра встречает титульником, позволяет поставить паузу, показывает итоги уровня и провожает титрами. Всё это — **экраны**, и в этой главе мы научимся ими управлять.

План такой: сначала статус-бар внизу экрана, как в Doom, — с лицом героя, которое морщится от боли. Потом машина состояний экранов, класс App. Потом титульник с живым огнём из PlayStation-версии Doom, пауза, итоги и титры. И напоследок — автокарта, которая открывается по мере того, как ты смотришь по сторонам.

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.