|  |
| --- |
| **boyar\_tournament/src/dev\_tools.rs** |
| use bevy::{  dev\_tools::{  fps\_overlay::FpsOverlayPlugin,  states::log\_transitions,  ui\_debug\_overlay::{DebugUiPlugin, UiDebugOptions},  },  input::common\_conditions::input\_just\_pressed,  prelude::\*, }; use bevy\_inspector\_egui::quick::WorldInspectorPlugin;  use crate::screens::GameState;  pub(super) fn plugin(app: &mut App) {  app.add\_systems(Update, log\_transitions::<GameState>);   app.add\_plugins(FpsOverlayPlugin::default());   app.add\_plugins(WorldInspectorPlugin::new());   app.add\_plugins(DebugUiPlugin);  app.add\_systems(  Update,  toggle\_debug\_ui.run\_if(input\_just\_pressed(KeyCode::Backquote)),  ); }  fn toggle\_debug\_ui(mut options: ResMut<UiDebugOptions>) {  options.toggle(); } |

|  |
| --- |
| **boyar\_tournament\src\lib.rs** |
| #[cfg(debug\_assertions)] mod dev\_tools; mod scaling; mod screens;  use bevy::{audio::Volume, prelude::\*};  pub struct GamePlugin;  impl Plugin for GamePlugin {  fn build(&self, app: &mut App) {  app.add\_systems(Startup, spawn\_camera);   app.insert\_resource(GlobalVolume {  volume: Volume::new(0.3),  });   app.add\_plugins((scaling::plugin, screens::plugin));   #[cfg(debug\_assertions)]  app.add\_plugins(dev\_tools::plugin);  } }  fn spawn\_camera(mut cmd: Commands) {  cmd.spawn((Camera2d, IsDefaultUiCamera)); } |

|  |
| --- |
| boyar\_tournament\src\scaling.rs |
| use bevy::{prelude::\*, window::WindowResized};  pub(super) fn plugin(app: &mut App) {  app.init\_resource::<DrawRegion>();  app.register\_type::<DrawRegion>();  app.register\_type::<DynamicScale>();  app.register\_type::<DynamicTransform>();   app.add\_systems(  PreUpdate,  (  update\_draw\_region,  update\_dynamic\_scale,  update\_dynamic\_transform,  )  .chain(),  );   #[cfg(debug\_assertions)]  app.add\_systems(Update, draw\_draw\_region\_outline); }  /// Регион 9x16(состоит из квадратов), внутри которого происходит вся отрисовка /// Длина и ширина его сторон определяют размер для всех сущностей #[derive(Resource, Reflect, Default)] #[reflect(Resource)] pub struct DrawRegion {  pub width: f32,  pub height: f32, }  fn update\_draw\_region(  mut draw\_region: ResMut<DrawRegion>,  mut resize\_events: EventReader<WindowResized>, ) {  if resize\_events.is\_empty() {  return;  }   for r\_e in resize\_events.read() {  let (aspect\_ratio\_width, aspect\_ratio\_height) = (9., 16.);  let (window\_width, window\_height) = (r\_e.width, r\_e.height);   // При длинном окне, DrawRegion по y на весь экран  if window\_height < window\_width / aspect\_ratio\_width \* aspect\_ratio\_height {  draw\_region.height = window\_height;  draw\_region.width = draw\_region.height / aspect\_ratio\_height \* aspect\_ratio\_width;  } else {  // При высоком окне, DrawRegion по x на весь экран  draw\_region.width = window\_width;  draw\_region.height = draw\_region.width / aspect\_ratio\_width \* aspect\_ratio\_height;  }  } }  /// Компонент для регулирования размеров Sprite /// Значение scale в компоненте Transform при размере окна игры 1920x1080 #[derive(Component, Reflect)] #[reflect(Component)] pub struct DynamicScale(pub f32);  fn update\_dynamic\_scale(  mut dynamic\_scale: Query<(&mut Transform, &DynamicScale)>,  draw\_region: Res<DrawRegion>, ) {  for (mut transform, dynamic\_scale) in &mut dynamic\_scale {  transform.scale = Vec3::splat(dynamic\_scale.0) \* draw\_region.height / 1080.;  } }  /// Расположение сущности в квадратах DrawRegion #[derive(Component, Reflect, Default)] #[reflect(Component)] pub struct DynamicTransform(pub f32, pub f32);  fn update\_dynamic\_transform(  mut dynamic\_transform: Query<(&mut Transform, &DynamicTransform)>,  draw\_region: Res<DrawRegion>, ) {  for (mut transform, dynamic\_transform) in &mut dynamic\_transform {  transform.translation.x = dynamic\_transform.0 \* draw\_region.width / 9.;  transform.translation.y = dynamic\_transform.1 \* draw\_region.height / 16.;  } }  #[cfg(debug\_assertions)] fn draw\_draw\_region\_outline(  mut toggle: Local<bool>,  keyboard: Res<ButtonInput<KeyCode>>,  mut gizmos: Gizmos,  draw\_region: Res<DrawRegion>, ) {  use bevy::math::vec2;   if keyboard.just\_pressed(KeyCode::F1) {  \*toggle ^= true;  }  if !\*toggle {  return;  }   gizmos  .grid\_2d(  Isometry2d::IDENTITY,  UVec2::new(9, 16),  vec2(draw\_region.width / 9., draw\_region.height / 16.),  Color::srgb(1., 0., 0.),  )  .outer\_edges(); } |

|  |
| --- |
| boyar\_tournament\src\screens\loading.rs |
| use bevy::prelude::\*; use bevy\_asset\_loader::prelude::\*;  use crate::scaling::DynamicScale;  use super::GameState;  pub(super) fn plugin(app: &mut App) {  app.add\_loading\_state(  LoadingState::new(GameState::Loading).continue\_to\_state(GameState::Gameplay),  );   app.add\_systems(OnEnter(GameState::Loading), spawn\_loading\_screen); }  fn spawn\_loading\_screen(mut cmd: Commands, asset\_server: Res<AssetServer>) {  cmd.spawn((  Sprite {  image: asset\_server.load("screens/loading/loading.png"),  ..default()  },  StateScoped(GameState::Loading),  DynamicScale(2.),  ));  cmd.spawn((  AudioPlayer::new(asset\_server.load("screens/loading/loading.ogg")),  PlaybackSettings::DESPAWN,  )); } |

|  |
| --- |
| boyar\_tournament\src\screens\mod.rs |
| use bevy::prelude::\*;  mod gameplay; mod loading; mod splash; mod ui;  pub(super) fn plugin(app: &mut App) {  app.init\_state::<GameState>();  app.enable\_state\_scoped\_entities::<GameState>();   app.add\_plugins((  splash::plugin,  loading::plugin,  gameplay::plugin,  ui::plugin,  )); }  #[derive(States, Debug, PartialEq, Eq, Clone, Hash, Default, Copy)] pub enum GameState {  #[default]  Splash,  Loading,  // Menu,  Gameplay, } |

|  |
| --- |
| boyar\_tournament\src\screens\splash.rs |
| use bevy::prelude::\*;  use crate::scaling::DynamicScale;  use super::GameState;  pub(super) fn plugin(app: &mut App) {  app.insert\_resource(ClearColor(Color::BLACK));  app.add\_systems(OnEnter(GameState::Splash), spawn\_splash\_screen);   app.add\_systems(  Update,  update\_fade\_in\_out.run\_if(in\_state(GameState::Splash)),  );   app.add\_systems(OnEnter(GameState::Splash), insert\_splash\_timer);  app.add\_systems(OnExit(GameState::Splash), remove\_splash\_timer);  app.add\_systems(  Update,  update\_splash\_timer.run\_if(in\_state(GameState::Splash)),  ); }  const SPLASH\_DURATION\_SEC: f32 = 1.5; const SPLASH\_FADE\_DURATION\_SEC: f32 = 0.75;  fn spawn\_splash\_screen(mut cmd: Commands, asset\_server: Res<AssetServer>) {  cmd.spawn((  Sprite {  image: asset\_server.load("screens/splash/valetoriy.png"),  ..default()  },  ImageFadeInOut {  total\_duration\_sec: SPLASH\_DURATION\_SEC,  fade\_duration\_sec: SPLASH\_FADE\_DURATION\_SEC,  t: 0.,  },  StateScoped(GameState::Splash),  DynamicScale(0.5),  ));  cmd.spawn((  AudioPlayer::new(asset\_server.load("screens/splash/splash.ogg")),  PlaybackSettings::DESPAWN,  )); }  #[derive(Component)] struct ImageFadeInOut {  total\_duration\_sec: f32,  fade\_duration\_sec: f32,  /// Текущий прогресс от 0 до total\_duration\_sec  t: f32, }  impl ImageFadeInOut {  fn alpha(&self) -> f32 {  let t = (self.t / self.total\_duration\_sec).clamp(0.0, 1.0);  let fade = self.fade\_duration\_sec / self.total\_duration\_sec;   // Трапезоидный график прозрачности  ((1.0 - (2.0 \* t - 1.0).abs()) / fade).min(1.0)  } }  fn update\_fade\_in\_out(  time: Res<Time>,  mut animation\_query: Query<(&mut ImageFadeInOut, &mut Sprite)>, ) {  let (mut anim, mut sprite) = animation\_query.single\_mut();  anim.t += time.delta\_secs();  sprite.color.set\_alpha(anim.alpha()) }  #[derive(Resource, Debug, Clone, PartialEq)] struct SplashTimer(Timer);  fn insert\_splash\_timer(mut cmd: Commands) {  cmd.insert\_resource(SplashTimer(Timer::from\_seconds(  SPLASH\_DURATION\_SEC,  TimerMode::Once,  ))); }  fn remove\_splash\_timer(mut cmd: Commands) {  cmd.remove\_resource::<SplashTimer>(); }  fn update\_splash\_timer(  time: Res<Time>,  mut timer: ResMut<SplashTimer>,  mut next\_screen: ResMut<NextState<GameState>>, ) {  timer.0.tick(time.delta());   if timer.0.just\_finished() {  next\_screen.set(GameState::Loading);  } } |

|  |
| --- |
| boyar\_tournament\src\screens\ui.rs |
| use bevy::{prelude::\*, window::PrimaryWindow};  use crate::scaling::{DrawRegion, DynamicTransform};  pub(super) fn plugin(app: &mut App) {  app.register\_type::<UiInteraction>();  app.register\_type::<UiHitbox>();   app.add\_systems(Update, (update\_ui\_hitboxes, trigger\_on\_press));   #[cfg(debug\_assertions)]  app.add\_systems(Update, draw\_ui\_hitboxes\_outline); }  #[derive(Component, Reflect, Default)] #[reflect(Component)] enum UiInteraction {  #[default]  None,  Hovered,  Pressed, }  #[derive(Component, Reflect)] #[reflect(Component)] #[require(UiInteraction, DynamicTransform)] // Длина и ширина прямоугольника хитбокса в клетках DynamicTransform pub struct UiHitbox(pub f32, pub f32);  fn update\_ui\_hitboxes(  mut query: Query<(&UiHitbox, &DynamicTransform, &mut UiInteraction)>,  window: Query<&Window, With<PrimaryWindow>>,  draw\_region: Res<DrawRegion>,  mouse: Res<ButtonInput<MouseButton>>,  touch: Res<Touches>, ) {  let window = window.single();  let mut press\_pos = if let Some(mouse\_pos) = window.cursor\_position() {  mouse\_pos  } else {  let Some(touch\_pos) = touch.first\_pressed\_position() else {  return;  };  touch\_pos  };  press\_pos.x -= window.width() / 2.;  press\_pos.y -= window.height() / 2.;  press\_pos.y \*= -1.;   let cell\_width = draw\_region.width / 9.;  let cell\_height = draw\_region.height / 16.;  for (hitbox, transform, mut interaction) in &mut query {  let hitbox\_bottom = (transform.1 - hitbox.1 / 2.) \* cell\_height;  let hitbox\_top = (transform.1 + hitbox.1 / 2.) \* cell\_height;  let hitbox\_left = (transform.0 - hitbox.0 / 2.) \* cell\_width;  let hitbox\_right = (transform.0 + hitbox.0 / 2.) \* cell\_width;   if hitbox\_bottom <= press\_pos.y  && press\_pos.y <= hitbox\_top  && hitbox\_left <= press\_pos.x  && press\_pos.x <= hitbox\_right  {  \*interaction = UiInteraction::Hovered;   if mouse.just\_pressed(MouseButton::Left) || touch.any\_just\_pressed() {  \*interaction = UiInteraction::Pressed;  }  continue;  }   \*interaction = UiInteraction::None;  } }  #[derive(Event)] pub struct OnPress;  fn trigger\_on\_press(  interaction\_query: Query<(Entity, &UiInteraction)>,  mut commands: Commands, ) {  for (entity, interaction) in &interaction\_query {  if matches!(interaction, UiInteraction::Pressed) {  commands.trigger\_targets(OnPress, entity);  }  } }  #[cfg(debug\_assertions)] fn draw\_ui\_hitboxes\_outline(  mut toggle: Local<bool>,  keyboard: Res<ButtonInput<KeyCode>>,  mut gizmos: Gizmos,  draw\_region: Res<DrawRegion>,  query: Query<(&UiHitbox, &DynamicTransform)>, ) {  use bevy::math::vec2;   if keyboard.just\_pressed(KeyCode::F3) {  \*toggle ^= true;  }  if !\*toggle {  return;  }   let cell\_width = draw\_region.width / 9.;  let cell\_height = draw\_region.height / 16.;  for (hitbox, transform) in &query {  gizmos.rect\_2d(  Isometry2d::from\_translation(vec2(  transform.0 \* cell\_width,  transform.1 \* cell\_height,  )),  vec2(hitbox.0 \* cell\_width, hitbox.1 \* cell\_height),  Color::srgb(0., 0., 1.),  );  } } |

|  |
| --- |
| boyar\_tournament\src\screens\gameplay\arena.rs |
| use bevy::prelude::\*; use bevy::window::PrimaryWindow; use bevy\_aseprite\_ultra::prelude::\*; use bevy\_asset\_loader::prelude::\*;  use crate::{scaling::DynamicScale, screens::GameState};  use crate::scaling::DrawRegion; use common::ArenaPos;  pub(super) fn plugin(app: &mut App) {  app.register\_type::<ArenaPos>();  app.register\_type::<ArenaHeightOffset>();  app.register\_type::<MouseArenaPos>();   app.init\_resource::<MouseArenaPos>();   app.configure\_loading\_state(  LoadingStateConfig::new(GameState::Loading).load\_collection::<ArenaAssets>(),  );   app.add\_systems(OnEnter(GameState::Gameplay), spawn\_arena);   app.add\_systems(  Update,  (update\_arena\_pos, update\_mouse\_arena\_pos).run\_if(in\_state(GameState::Gameplay)),  );   #[cfg(debug\_assertions)]  app.add\_systems(  Update,  draw\_arena\_region\_outline.run\_if(in\_state(GameState::Gameplay)),  ); }  #[derive(AssetCollection, Resource)] struct ArenaAssets {  #[asset(path = "arena/winter\_arena.aseprite")]  arena: Handle<Aseprite>,  #[asset(path = "arena/battle.ogg")]  battle\_music: Handle<AudioSource>, }  fn spawn\_arena(mut cmd: Commands, arena\_assets: ResMut<ArenaAssets>) {  cmd.spawn((  Name::new("Шаблон арены"),  AseSpriteSlice {  name: "winter\_arena".into(),  aseprite: arena\_assets.arena.clone(),  },  StateScoped(GameState::Gameplay),  DynamicScale(1.),  Transform::from\_translation(Vec3::ZERO.with\_z(-0.5)),  ));  cmd.spawn((  AudioPlayer::new(arena\_assets.battle\_music.clone()),  PlaybackSettings::LOOP,  StateScoped(GameState::Gameplay),  )); }  #[derive(Component, Reflect, Clone, Copy)] #[reflect(Component)] pub struct ArenaHeightOffset(pub f32);  fn update\_arena\_pos(  mut arena\_pos: Query<(&mut Transform, &ArenaPos, Option<&ArenaHeightOffset>)>,  draw\_region: Res<DrawRegion>, ) {  for (mut transform, arena\_pos, height\_offset) in &mut arena\_pos {  transform.translation.x = arena\_pos.0 \* draw\_region.width / 19.61;  transform.translation.y =  arena\_pos.1 \* draw\_region.height / 43.2 + draw\_region.height / 13.5;   // Чем ниже сущность на арене тем "выше" она отображается  transform.translation.z = transform.translation.y / draw\_region.height \* -1.;   if let Some(height\_offset) = height\_offset {  transform.translation.y += height\_offset.0 \* draw\_region.height / 43.2;  }  } }  #[derive(Resource, Reflect, Default)] #[reflect(Resource)] pub struct MouseArenaPos(pub Option<ArenaPos>);  fn update\_mouse\_arena\_pos(  mut mouse\_arena\_pos: ResMut<MouseArenaPos>,  window: Query<&Window, With<PrimaryWindow>>,  draw\_region: Res<DrawRegion>,  touch: Res<Touches>, ) {  let window = window.single();  let mut press\_pos = if let Some(mouse\_pos) = window.cursor\_position() {  mouse\_pos  } else {  let Some(touch\_pos) = touch.first\_pressed\_position() else {  return;  };  touch\_pos  };  press\_pos.x -= window.width() / 2.;  press\_pos.y -= window.height() / 2.;  press\_pos.y \*= -1.;   press\_pos.y -= draw\_region.height / 13.5;  press\_pos.x /= draw\_region.width / 19.61;  press\_pos.y /= draw\_region.height / 43.2;  if press\_pos.x.abs() <= 9. && press\_pos.y.abs() <= 16. {  mouse\_arena\_pos.0 = Some(ArenaPos(press\_pos.x, press\_pos.y));  return;  }   mouse\_arena\_pos.0 = None; }  #[cfg(debug\_assertions)] fn draw\_arena\_region\_outline(  mut toggle: Local<bool>,  keyboard: Res<ButtonInput<KeyCode>>,  mut gizmos: Gizmos,  draw\_region: Res<DrawRegion>, ) {  use bevy::math::vec2;   if keyboard.just\_pressed(KeyCode::F2) {  \*toggle ^= true;  }  if !\*toggle {  return;  }   gizmos  .grid\_2d(  Isometry2d::from\_translation(vec2(0., draw\_region.height / 13.5)),  UVec2::new(18, 32),  vec2(draw\_region.width / 19.61, draw\_region.height / 43.2),  Color::srgb(1., 0.65, 0.),  )  .outer\_edges(); } |

|  |
| --- |
| boyar\_tournament\src\screens\gameplay\deck.rs |
| use bevy::{input::common\_conditions::input\_just\_released, prelude::\*}; use bevy\_aseprite\_ultra::prelude::\*; use bevy\_asset\_loader::prelude::\*; use bevy\_quinnet::client::QuinnetClient; use common::{ArenaPos, Card, ClientChannel, ClientMessage, PlayerNumber}; use rand::{seq::SliceRandom, thread\_rng};  use crate::{  scaling::{DynamicScale, DynamicTransform},  screens::{  ui::{OnPress, UiHitbox},  GameState,  }, };  use super::{arena::MouseArenaPos, spawn\_text, FontAssets};  pub(super) fn plugin(app: &mut App) {  app.register\_type::<Deck>();  app.register\_type::<DeckIndex>();  app.register\_type::<SelectedCard>();  app.register\_type::<ElixirCounter>();   app.init\_resource::<SelectedCard>();  app.init\_resource::<ElixirCounter>();   app.configure\_loading\_state(  LoadingStateConfig::new(GameState::Loading).load\_collection::<CardsAssets>(),  );   use Card::\*;  let mut cards = [  Rus,  Musketeer,  ThreeMusketeers,  Priest,  Bats,  BatHorde,  Bomber,  Giant,  ];  cards.shuffle(&mut thread\_rng());  app.insert\_resource(Deck(cards));   app.add\_systems(  Update,  play\_card.run\_if(  in\_state(GameState::Gameplay).and(  input\_just\_released(MouseButton::Left)  .or(|touch: Res<Touches>| touch.any\_just\_released()),  ),  ),  );  app.add\_systems(  Update,  update\_elixir\_counter.run\_if(in\_state(GameState::Gameplay)),  );   app.add\_systems(  OnEnter(GameState::Gameplay),  (spawn\_card\_hand, spawn\_elixir\_counter),  );  app.add\_observer(update\_card\_hand); }  #[derive(AssetCollection, Resource)] struct CardsAssets {  #[asset(path = "cards.aseprite")]  cards: Handle<Aseprite>,  #[asset(path = "screens/gameplay/card\_select.ogg")]  card\_select: Handle<AudioSource>, }  #[derive(Resource, Reflect)] #[reflect(Resource)] struct Deck([Card; 8]);  #[derive(Component, Reflect, Clone, Copy)] #[reflect(Component)] struct DeckIndex(u8);  fn spawn\_card\_hand(  mut cmd: Commands,  cards\_assets: ResMut<CardsAssets>,  deck: Res<Deck>,  font: Res<FontAssets>, ) {  for (i, (pos, card)) in [-2.05, -0.22, 1.62, 3.45].iter().zip(deck.0).enumerate() {  cmd.spawn((  Name::new(format!("Карта {}", i + 1)),  AseSpriteSlice {  name: card.tag(),  aseprite: cards\_assets.cards.clone(),  },  DeckIndex(i as \_),  StateScoped(GameState::Gameplay),  DynamicScale(1.8),  DynamicTransform(\*pos, -6.279),  UiHitbox(1.8, 2.3),  ))  .observe(on\_card\_select);  }   spawn\_text(  &mut cmd,  "След.",  font.font.clone(),  35.,  Color::srgb(1., 1., 0.),  1.,  (-3.8, -5.05),  GameState::Gameplay,  );  cmd.spawn((  Name::new("Следующая карта"),  AseSpriteSlice {  name: deck.0[4].tag(),  aseprite: cards\_assets.cards.clone(),  },  DeckIndex(4),  StateScoped(GameState::Gameplay),  DynamicScale(0.8),  DynamicTransform(-3.8, -5.7),  )); }  #[derive(Resource, Reflect)] #[reflect(Resource)] struct ElixirCounter(u8, Timer); impl Default for ElixirCounter {  fn default() -> Self {  Self(0, Timer::from\_seconds(1.5, TimerMode::Repeating))  } }  fn spawn\_elixir\_counter(mut cmd: Commands, font: Res<FontAssets>) {  spawn\_text(  &mut cmd,  "0",  font.font.clone(),  35.,  Color::srgb(1., 0., 1.),  1.,  (0.7, -7.7),  GameState::Gameplay,  ); }  fn update\_elixir\_counter(  mut counter: ResMut<ElixirCounter>,  mut text: Query<&mut Text2d>,  time: Res<Time>, ) {  if counter.1.tick(time.delta()).just\_finished() {  if counter.0 < 10 {  counter.0 += 1;  }  }   for mut text in &mut text {  if text.0 != "След." {  text.0 = counter.0.to\_string();  }  } }  trait IntoTag {  fn tag(&self) -> String; } impl IntoTag for Card {  fn tag(&self) -> String {  let s = match self {  Card::Musketeer => "musketeer",  Card::Rus => "rus",  Card::ThreeMusketeers => "three\_musketeers",  Card::Priest => "priest",  Card::Bats => "bats",  Card::BatHorde => "bat\_horde",  Card::Bomber => "bomber",  Card::Giant => "giant",  };  s.into()  } } trait ElixirCost {  fn elixir\_cost(&self) -> u8; } impl ElixirCost for Card {  fn elixir\_cost(&self) -> u8 {  match self {  Card::Rus => 3,  Card::Musketeer => 4,  Card::ThreeMusketeers => 9,  Card::Priest => 5,  Card::Bats => 3,  Card::BatHorde => 5,  Card::Bomber => 3,  Card::Giant => 6,  }  } }  #[derive(Resource, Reflect, Default)] #[reflect(Resource)] struct SelectedCard(Option<u8>);  const SELECTED\_CARD\_SCALE\_AMOUNT: f32 = 0.2;  fn on\_card\_select(  trigger: Trigger<OnPress>,  mut selected\_card: ResMut<SelectedCard>,  mut query: Query<(&DeckIndex, &mut DynamicScale)>,  mut cmd: Commands,  cards\_assets: ResMut<CardsAssets>, ) {  cmd.spawn((  AudioPlayer::new(cards\_assets.card\_select.clone()),  PlaybackSettings::DESPAWN,  ));   let entity = trigger.entity();  let (&pressed\_index, \_) = query.get(entity).unwrap();   if let Some(selected\_index) = selected\_card.0 {  for (index, mut scale) in &mut query {  if index.0 == selected\_index {  scale.0 -= SELECTED\_CARD\_SCALE\_AMOUNT;  selected\_card.0 = None;   if index.0 == pressed\_index.0 {  return;  }  }  }  }   let (\_, mut pressed\_scale) = query.get\_mut(entity).unwrap();  selected\_card.0 = Some(pressed\_index.0);  pressed\_scale.0 += SELECTED\_CARD\_SCALE\_AMOUNT; }  fn play\_card(  mouse\_pos: Res<MouseArenaPos>,  selected\_card: Res<SelectedCard>,  mut deck: ResMut<Deck>,  mut client: ResMut<QuinnetClient>,  mut cmd: Commands,  player\_num: Res<PlayerNumber>,  mut elixir: ResMut<ElixirCounter>, ) {  let Some(mouse\_pos) = mouse\_pos.0 else {  return;  };  let Some(index) = selected\_card.0 else {  return;  };  let index = index as usize;  let card = deck.0[index];   let cost = card.elixir\_cost();  if cost > elixir.0 {  return;  }  elixir.0 -= cost;   // Ставим точку в центр клетки  let mut x = mouse\_pos.0.floor() + 0.5;  let mut y = mouse\_pos.1.floor().clamp(-16., -2.) + 0.5;  if let PlayerNumber::Two = \*player\_num {  x \*= -1.;  y \*= -1.;  }   client  .connection\_mut()  .send\_message\_on(  ClientChannel::OrderedReliable,  ClientMessage::PlayCard {  card,  placement: ArenaPos(x, y),  },  )  .unwrap();   // Передвигаем карты в колоде на 1  deck.0[index] = deck.0[4];  deck.0[4] = deck.0[5];  deck.0[5] = deck.0[6];  deck.0[6] = deck.0[7];  deck.0[7] = card;   cmd.trigger(UpdateCardHand); }  #[derive(Event)] struct UpdateCardHand;  fn update\_card\_hand(  \_: Trigger<UpdateCardHand>,  deck: Res<Deck>,  mut query: Query<(&DeckIndex, &mut AseSpriteSlice, &mut DynamicScale)>,  mut selected\_card: ResMut<SelectedCard>, ) {  for (index, mut sprite, mut scale) in &mut query {  if index.0 == selected\_card.0.unwrap() {  scale.0 -= SELECTED\_CARD\_SCALE\_AMOUNT;  }   let card = deck.0[index.0 as usize];  sprite.name = card.tag();  }   selected\_card.0 = None; } boyar\_tournament\src\screens\gameplay\mod.rs use bevy::prelude::\*; use bevy\_aseprite\_ultra::prelude::\*; use bevy\_asset\_loader::prelude::\*;  use crate::scaling::{DynamicScale, DynamicTransform};  use super::GameState;  mod arena; mod deck; mod networking; mod projectiles; mod units;  pub(super) fn plugin(app: &mut App) {  app.add\_plugins(AsepriteUltraPlugin);   app.add\_plugins((  arena::plugin,  networking::plugin,  units::plugin,  deck::plugin,  projectiles::plugin,  ));   app.configure\_loading\_state(  LoadingStateConfig::new(GameState::Loading).load\_collection::<FontAssets>(),  ); }  #[derive(AssetCollection, Resource)] struct FontAssets {  #[asset(path = "Keleti-Regular.ttf")]  font: Handle<Font>, }  fn spawn\_text(  cmd: &mut Commands,  text: &str,  font: Handle<Font>,  font\_size: f32,  color: Color,  dynamic\_scale: f32,  dynamic\_transform: (f32, f32),  state: GameState, ) {  cmd.spawn((  Text2d::new(text),  TextFont::from\_font(font.clone()).with\_font\_size(font\_size),  TextColor(color),  StateScoped(state),  DynamicScale(dynamic\_scale),  DynamicTransform(dynamic\_transform.0, dynamic\_transform.1),  ))  .insert(Transform::from\_xyz(0., 0., 0.2));   cmd.spawn((  Text2d::new(text),  TextFont::from\_font(font.clone()).with\_font\_size(font\_size),  TextColor(Color::BLACK),  StateScoped(state),  DynamicScale(dynamic\_scale),  DynamicTransform(dynamic\_transform.0 + 0.03, dynamic\_transform.1 - 0.03),  ))  .insert(Transform::from\_xyz(0., 0., 0.1)); } |

|  |
| --- |
| boyar\_tournament\src\screens\gameplay\networking.rs |
| use bevy::{prelude::\*, utils::HashMap}; use bevy\_quinnet::client::{  certificate::CertificateVerificationMode, connection::ClientEndpointConfiguration,  QuinnetClient, QuinnetClientPlugin, }; use common::{  ArenaPos, ClientChannel, Direction, Health, PlayerNumber, ServerMessage, UnitState,  LOCAL\_BIND\_IP, SERVER\_HOST, SERVER\_PORT, };  use crate::screens::GameState;  use super::{  projectiles::SpawnProjectile,  units::{AssociatedTower, SpawnUnit}, };  pub(super) fn plugin(app: &mut App) {  app.add\_plugins(QuinnetClientPlugin::default());   app.init\_resource::<PlayerNumber>();  app.init\_resource::<NetworkMapping>();  app.register\_type::<NetworkMapping>();   app.add\_systems(OnEnter(GameState::Gameplay), start\_connection);  app.add\_systems(  Update,  handle\_server\_messages.run\_if(in\_state(GameState::Gameplay)),  ); }  fn start\_connection(mut client: ResMut<QuinnetClient>) {  client  .open\_connection(  ClientEndpointConfiguration::from\_ips(SERVER\_HOST, SERVER\_PORT, LOCAL\_BIND\_IP, 0),  CertificateVerificationMode::SkipVerification,  ClientChannel::channels\_config(),  )  .unwrap(); }  trait AdjustForPlayer {  fn adjust\_for\_player(&self, player\_num: PlayerNumber) -> Self; } impl AdjustForPlayer for ArenaPos {  fn adjust\_for\_player(&self, player\_num: PlayerNumber) -> Self {  match player\_num {  PlayerNumber::One => \*self,  PlayerNumber::Two => ArenaPos(-self.0, -self.1),  }  } } impl AdjustForPlayer for Direction {  fn adjust\_for\_player(&self, player\_num: PlayerNumber) -> Self {  match player\_num {  PlayerNumber::One => \*self,  PlayerNumber::Two => self.opposite(),  }  } }  fn handle\_server\_messages(  mut client: ResMut<QuinnetClient>,  mut player\_num: ResMut<PlayerNumber>,  mut cmd: Commands,  mut network\_mapping: ResMut<NetworkMapping>,  mut units\_query: Query<(&mut ArenaPos, &mut Direction, &mut UnitState, &mut Health)>,  mut projectiles\_query: Query<&mut ArenaPos, Without<UnitState>>,  towers: Query<&AssociatedTower>, ) {  while let Some((\_, message)) = client  .connection\_mut()  .try\_receive\_message::<ServerMessage>()  {  match message {  ServerMessage::StartGame(n) => \*player\_num = n,  ServerMessage::SpawnUnit {  server\_entity,  unit,  pos,  owner,  } => {  unit.spawn(  server\_entity,  pos.adjust\_for\_player(\*player\_num),  owner,  &mut cmd,  );  }  ServerMessage::SpawnProjectile {  server\_entity,  projectile,  attacker,  receiver,  pos,  } => projectile.spawn(  server\_entity,  attacker,  receiver,  pos.adjust\_for\_player(\*player\_num),  &mut cmd,  ),  ServerMessage::Despawn(server\_entity) => {  let Some(entity) = network\_mapping.remove(&server\_entity) else {  continue;  };  if let Ok(tower) = towers.get(entity) {  cmd.entity(tower.0).despawn();  }  cmd.entity(entity).despawn();  }  ServerMessage::SyncEntities { units, projectiles } => {  for (server\_entity, pos, direction, state, health) in &units {  let Some(&entity) = network\_mapping.get(server\_entity) else {  continue;  };  let (mut p, mut d, mut s, mut h) = units\_query.get\_mut(entity).unwrap();  \*p = pos.adjust\_for\_player(\*player\_num);  \*d = direction.adjust\_for\_player(\*player\_num);  \*s = \*state;  \*h = \*health;  }   for (server\_entity, pos) in &projectiles {  let Some(&entity) = network\_mapping.get(server\_entity) else {  continue;  };  let mut p = projectiles\_query.get\_mut(entity).unwrap();  \*p = pos.adjust\_for\_player(\*player\_num);  }  }  }  } }  #[derive(Resource, Reflect, Default, Deref, DerefMut)] #[reflect(Resource)] // Сопоставление Entity сервера и клиента pub struct NetworkMapping(HashMap<Entity, Entity>); |

|  |
| --- |
| boyar\_tournament\src\screens\gameplay\projectiles\bomb.rs |
| use bevy::prelude::\*; use bevy\_aseprite\_ultra::prelude::\*; use bevy\_asset\_loader::prelude::\*; use common::ArenaPos;  use crate::{  scaling::DynamicScale,  screens::{  gameplay::{arena::ArenaHeightOffset, networking::NetworkMapping},  GameState,  }, };  use super::ProjectileTargets;  pub(super) fn plugin(app: &mut App) {  app.add\_observer(spawn\_bomb);   app.configure\_loading\_state(  LoadingStateConfig::new(GameState::Loading).load\_collection::<BombAssets>(),  ); }  #[derive(Event)] pub struct SpawnBomb(pub Entity, pub Entity, pub Entity, pub ArenaPos);  #[derive(Component)] #[require(  Name(|| Name::new("Бомба")),  DynamicScale(|| DynamicScale(0.5)),  ArenaHeightOffset(|| ArenaHeightOffset(0.3)), )] struct Bomb;  #[derive(Resource, AssetCollection)] struct BombAssets {  #[asset(path = "units/bomber/bomb.aseprite")]  sprite: Handle<Aseprite>, }  fn spawn\_bomb(  trigger: Trigger<SpawnBomb>,  mut cmd: Commands,  assets: ResMut<BombAssets>,  mut network\_mapping: ResMut<NetworkMapping>, ) {  let &SpawnBomb(entity, attacker, receiver, pos) = trigger.event();  let Some(attacker) = network\_mapping.get(&attacker) else {  return;  };  let Some(receiver) = network\_mapping.get(&receiver) else {  return;  };   let bomb = cmd  .spawn((  Bomb,  pos,  AseSpriteAnimation {  animation: Animation::tag("bomb"),  aseprite: assets.sprite.clone(),  },  ProjectileTargets(\*attacker, \*receiver, 0.5),  ))  .id();   network\_mapping.insert(entity, bomb); } |

|  |
| --- |
| boyar\_tournament\src\screens\gameplay\projectiles\bullet.rs |
| use bevy::prelude::\*; use bevy\_aseprite\_ultra::prelude::\*; use bevy\_asset\_loader::prelude::\*; use common::ArenaPos;  use crate::{  scaling::DynamicScale,  screens::{  gameplay::{arena::ArenaHeightOffset, networking::NetworkMapping},  GameState,  }, };  use super::ProjectileTargets;  pub(super) fn plugin(app: &mut App) {  app.add\_observer(spawn\_bullet);   app.configure\_loading\_state(  LoadingStateConfig::new(GameState::Loading).load\_collection::<BulletAssets>(),  ); }  #[derive(Event)] pub struct SpawnBullet(pub Entity, pub Entity, pub Entity, pub ArenaPos);  #[derive(Component)] #[require(  Name(|| Name::new("Пуля")),  DynamicScale(|| DynamicScale(1.)),  ArenaHeightOffset(|| ArenaHeightOffset(0.)), )] struct Bullet;  #[derive(Resource, AssetCollection)] struct BulletAssets {  #[asset(path = "units/musketeer/bullet.aseprite")]  sprite: Handle<Aseprite>, }  fn spawn\_bullet(  trigger: Trigger<SpawnBullet>,  mut cmd: Commands,  assets: ResMut<BulletAssets>,  mut network\_mapping: ResMut<NetworkMapping>, ) {  let &SpawnBullet(entity, attacker, receiver, pos) = trigger.event();  let Some(attacker) = network\_mapping.get(&attacker) else {  return;  };  let Some(receiver) = network\_mapping.get(&receiver) else {  return;  };   let bullet = cmd  .spawn((  Bullet,  pos,  AseSpriteSlice {  name: "bullet".into(),  aseprite: assets.sprite.clone(),  },  ProjectileTargets(\*attacker, \*receiver, 0.5),  ))  .id();   network\_mapping.insert(entity, bullet); } |

|  |
| --- |
| boyar\_tournament\src\screens\gameplay\projectiles\fireball.rs |
| use bevy::prelude::\*; use bevy\_aseprite\_ultra::prelude::\*; use bevy\_asset\_loader::prelude::\*; use common::ArenaPos;  use crate::{  scaling::DynamicScale,  screens::{  gameplay::{arena::ArenaHeightOffset, networking::NetworkMapping},  GameState,  }, };  use super::ProjectileTargets;  pub(super) fn plugin(app: &mut App) {  app.add\_observer(spawn\_fireball);   app.configure\_loading\_state(  LoadingStateConfig::new(GameState::Loading).load\_collection::<FireballAssets>(),  ); }  #[derive(Event)] pub struct SpawnFireball(pub Entity, pub Entity, pub Entity, pub ArenaPos);  #[derive(Component)] #[require(  Name(|| Name::new("Фаерболл")),  DynamicScale(|| DynamicScale(0.5)),  ArenaHeightOffset(|| ArenaHeightOffset(0.)), )] struct Fireball;  #[derive(Resource, AssetCollection)] struct FireballAssets {  #[asset(path = "units/priest/fireball.aseprite")]  sprite: Handle<Aseprite>, }  fn spawn\_fireball(  trigger: Trigger<SpawnFireball>,  mut cmd: Commands,  assets: ResMut<FireballAssets>,  mut network\_mapping: ResMut<NetworkMapping>, ) {  let &SpawnFireball(entity, attacker, receiver, pos) = trigger.event();  let Some(attacker) = network\_mapping.get(&attacker) else {  return;  };  let Some(receiver) = network\_mapping.get(&receiver) else {  return;  };   let fireball = cmd  .spawn((  Fireball,  pos,  AseSpriteAnimation {  animation: Animation::tag("fireball"),  aseprite: assets.sprite.clone(),  },  ProjectileTargets(\*attacker, \*receiver, 0.5),  ))  .id();   network\_mapping.insert(entity, fireball); } |

|  |
| --- |
| boyar\_tournament\src\screens\gameplay\projectiles\mod.rs |
| use bevy::prelude::\*; use bomb::SpawnBomb; use bullet::SpawnBullet; use common::{ArenaPos, Projectile}; use fireball::SpawnFireball;  use crate::screens::GameState;  use super::arena::ArenaHeightOffset;  mod bomb; mod bullet; mod fireball;  pub(super) fn plugin(app: &mut App) {  app.add\_plugins((bullet::plugin, fireball::plugin, bomb::plugin));   app.add\_systems(  Update,  update\_projectile\_height.run\_if(in\_state(GameState::Gameplay)),  ); }  pub(super) trait SpawnProjectile {  fn spawn(  &self,  entity: Entity,  attacker: Entity,  receiver: Entity,  pos: ArenaPos,  cmd: &mut Commands,  ); }  impl SpawnProjectile for Projectile {  fn spawn(  &self,  entity: Entity,  attacker: Entity,  receiver: Entity,  pos: ArenaPos,  cmd: &mut Commands,  ) {  match self {  Projectile::Bullet => cmd.trigger(SpawnBullet(entity, attacker, receiver, pos)),  Projectile::Fireball => {  cmd.trigger(SpawnFireball(entity, attacker, receiver, pos))  }  Projectile::Bomb => cmd.trigger(SpawnBomb(entity, attacker, receiver, pos)),  }  } }  #[derive(Component)] struct ProjectileTargets(Entity, Entity, f32);  fn update\_projectile\_height(  projectiles: Query<(Entity, &ProjectileTargets)>,  mut positions: Query<(&ArenaPos, &mut ArenaHeightOffset)>, ) {  for (entity, targets) in &projectiles {  let Ok((&attacker\_pos, &attacker\_height)) = positions.get(targets.0) else {  continue;  };  let Ok((&receiver\_pos, &receiver\_height)) = positions.get(targets.1) else {  continue;  };  let Ok((self\_pos, mut self\_height)) = positions.get\_mut(entity) else {  continue;  };   let dist\_to\_attacker = self\_pos.distance(&attacker\_pos);  let dist\_to\_receiver = self\_pos.distance(&receiver\_pos);  let progress = dist\_to\_attacker / (dist\_to\_attacker + dist\_to\_receiver);   let height = attacker\_height.0 + progress \* (receiver\_height.0 - attacker\_height.0);  self\_height.0 = height + targets.2;  } } |

|  |
| --- |
| boyar\_tournament\src\screens\gameplay\units\archer\_tower.rs |
| use bevy::prelude::\*; use bevy\_aseprite\_ultra::prelude::\*; use bevy\_asset\_loader::prelude::\*; use common::{ArenaPos, Health, PlayerNumber, UnitState};  use crate::{  scaling::DynamicScale,  screens::{  gameplay::{arena::ArenaHeightOffset, networking::NetworkMapping},  GameState,  }, };  use super::{AssociatedTower, IntoTag, SpawnDirection};  pub(super) fn plugin(app: &mut App) {  app.add\_observer(spawn\_archer\_tower);   app.configure\_loading\_state(  LoadingStateConfig::new(GameState::Loading).load\_collection::<ArcherTowerAssets>(),  );   app.add\_systems(OnExit(GameState::Gameplay), despawn\_archer\_towers); }  #[derive(Event)] pub struct SpawnArcherTower(pub Entity, pub ArenaPos, pub PlayerNumber);  #[derive(Component)] #[require(  DynamicScale(|| DynamicScale(0.55)),  ArenaHeightOffset(|| ArenaHeightOffset(1.)), )] struct ArcherTower;  #[derive(Component)] #[require(  Health,   Name(|| Name::new("Лучник на башне")),  DynamicScale(|| DynamicScale(0.55)),  UnitState,  ArenaHeightOffset(|| ArenaHeightOffset(3.1)), )] struct ArcherTowerArcher;  #[derive(Resource, AssetCollection)] struct ArcherTowerAssets {  #[asset(path = "units/archer\_tower/ally\_tower.aseprite")]  ally\_tower: Handle<Aseprite>,  #[asset(path = "units/archer\_tower/enemy\_tower.aseprite")]  enemy\_tower: Handle<Aseprite>,   #[asset(path = "units/musketeer/ally\_musketeer.aseprite")]  ally\_archer: Handle<Aseprite>,  #[asset(path = "units/musketeer/enemy\_musketeer.aseprite")]  enemy\_archer: Handle<Aseprite>, }  fn spawn\_archer\_tower(  trigger: Trigger<SpawnArcherTower>,  mut cmd: Commands,  self\_num: Res<PlayerNumber>,  assets: ResMut<ArcherTowerAssets>,  mut network\_mapping: ResMut<NetworkMapping>, ) {  let SpawnArcherTower(entity, mut pos, player\_num) = trigger.event();   let direction = self\_num.spawn\_direction(\*player\_num);   let (tower\_sprite, archer\_sprite) = if pos.1 < 0. {  (assets.ally\_tower.clone(), assets.ally\_archer.clone())  } else {  (assets.enemy\_tower.clone(), assets.enemy\_archer.clone())  };   pos.1 += 0.01;  let tower = cmd  .spawn((  ArcherTower,  pos,  AseSpriteSlice {  name: "tower".into(),  aseprite: tower\_sprite,  },  ))  .id();   pos.1 -= 0.01;  let archer = cmd  .spawn((  ArcherTowerArcher,  pos,  direction,  AseSpriteAnimation {  animation: Animation::tag(direction.tag()),  aseprite: archer\_sprite,  },  AssociatedTower(tower),  ))  .id();  network\_mapping.insert(\*entity, archer); }  fn despawn\_archer\_towers(mut cmd: Commands, towers: Query<(Entity, &AssociatedTower)>) {  for (archer, tower) in towers.iter() {  cmd.entity(tower.0).despawn();  cmd.entity(archer).despawn();  } } |

|  |
| --- |
| boyar\_tournament\src\screens\gameplay\units\bat.rs |
| use bevy::prelude::\*; use bevy\_aseprite\_ultra::prelude::\*; use bevy\_asset\_loader::prelude::\*; use common::{ArenaPos, Health, PlayerNumber, UnitState};  use crate::{  scaling::DynamicScale,  screens::{  gameplay::{arena::ArenaHeightOffset, networking::NetworkMapping},  GameState,  }, };  use super::{IntoTag, SpawnDirection};  pub(super) fn plugin(app: &mut App) {  app.add\_observer(spawn\_bat);   app.configure\_loading\_state(  LoadingStateConfig::new(GameState::Loading).load\_collection::<BatAssets>(),  ); }  #[derive(Event)] pub struct SpawnBat(pub Entity, pub ArenaPos, pub PlayerNumber);  #[derive(Component)] #[require(  Health,   Name(|| Name::new("Мышь")),  DynamicScale(|| DynamicScale(0.3)),  UnitState,  ArenaHeightOffset(|| ArenaHeightOffset(2.5)), )] struct Bat;  #[derive(Resource, AssetCollection)] struct BatAssets {  #[asset(path = "units/bat/bat.aseprite")]  sprite: Handle<Aseprite>, }  fn spawn\_bat(  trigger: Trigger<SpawnBat>,  mut cmd: Commands,  self\_num: Res<PlayerNumber>,  assets: ResMut<BatAssets>,  mut network\_mapping: ResMut<NetworkMapping>, ) {  let &SpawnBat(entity, pos, player\_num) = trigger.event();   let direction = self\_num.spawn\_direction(player\_num);   let bat = cmd  .spawn((  Bat,  pos,  direction,  AseSpriteAnimation {  animation: Animation::tag(direction.tag()),  aseprite: assets.sprite.clone(),  },  ))  .id();   network\_mapping.insert(entity, bat); } |

|  |
| --- |
| boyar\_tournament\src\screens\gameplay\units\bomber.rs |
| use bevy::prelude::\*; use bevy\_aseprite\_ultra::prelude::\*; use bevy\_asset\_loader::prelude::\*; use common::{ArenaPos, Health, PlayerNumber, UnitState};  use crate::{  scaling::DynamicScale,  screens::{  gameplay::{arena::ArenaHeightOffset, networking::NetworkMapping},  GameState,  }, };  use super::{IntoTag, SpawnDirection};  pub(super) fn plugin(app: &mut App) {  app.add\_observer(spawn\_bomber);   app.configure\_loading\_state(  LoadingStateConfig::new(GameState::Loading).load\_collection::<BomberAssets>(),  ); }  #[derive(Event)] pub struct SpawnBomber(pub Entity, pub ArenaPos, pub PlayerNumber);  #[derive(Component)] #[require(  Health,   Name(|| Name::new("Подрывник")),  DynamicScale(|| DynamicScale(0.7)),  UnitState,  ArenaHeightOffset(|| ArenaHeightOffset(2.3)), )] struct Bomber;  #[derive(Resource, AssetCollection)] struct BomberAssets {  #[asset(path = "units/bomber/bomber.aseprite")]  sprite: Handle<Aseprite>, }  fn spawn\_bomber(  trigger: Trigger<SpawnBomber>,  mut cmd: Commands,  self\_num: Res<PlayerNumber>,  assets: ResMut<BomberAssets>,  mut network\_mapping: ResMut<NetworkMapping>, ) {  let &SpawnBomber(entity, pos, player\_num) = trigger.event();   let direction = self\_num.spawn\_direction(player\_num);   let bomber = cmd  .spawn((  Bomber,  pos,  direction,  AseSpriteAnimation {  animation: Animation::tag(direction.tag()),  aseprite: assets.sprite.clone(),  },  ))  .id();   network\_mapping.insert(entity, bomber); } |

|  |
| --- |
| boyar\_tournament\src\screens\gameplay\units\giant.rs |
| use bevy::prelude::\*; use bevy\_aseprite\_ultra::prelude::\*; use bevy\_asset\_loader::prelude::\*; use common::{ArenaPos, Health, PlayerNumber, UnitState};  use crate::{  scaling::DynamicScale,  screens::{  gameplay::{arena::ArenaHeightOffset, networking::NetworkMapping},  GameState,  }, };  use super::{IntoTag, SpawnDirection};  pub(super) fn plugin(app: &mut App) {  app.add\_observer(spawn\_giant);   app.configure\_loading\_state(  LoadingStateConfig::new(GameState::Loading).load\_collection::<GiantAssets>(),  ); }  #[derive(Event)] pub struct SpawnGiant(pub Entity, pub ArenaPos, pub PlayerNumber);  #[derive(Component)] #[require(  Health,   Name(|| Name::new("Гигант")),  DynamicScale(|| DynamicScale(1.)),  UnitState,  ArenaHeightOffset(|| ArenaHeightOffset(3.)), )] struct Giant;  #[derive(Resource, AssetCollection)] struct GiantAssets {  #[asset(path = "units/giant/giant.aseprite")]  sprite: Handle<Aseprite>, }  fn spawn\_giant(  trigger: Trigger<SpawnGiant>,  mut cmd: Commands,  self\_num: Res<PlayerNumber>,  assets: ResMut<GiantAssets>,  mut network\_mapping: ResMut<NetworkMapping>, ) {  let &SpawnGiant(entity, pos, player\_num) = trigger.event();   let direction = self\_num.spawn\_direction(player\_num);   let giant = cmd  .spawn((  Giant,  pos,  direction,  AseSpriteAnimation {  animation: Animation::tag(direction.tag()),  aseprite: assets.sprite.clone(),  },  ))  .id();   network\_mapping.insert(entity, giant); } |

|  |
| --- |
| boyar\_tournament\src\screens\gameplay\units\king\_tower.rs |
| use bevy::prelude::\*; use bevy\_aseprite\_ultra::prelude::\*; use bevy\_asset\_loader::prelude::\*; use common::{ArenaPos, Health, PlayerNumber, UnitState};  use crate::{  scaling::DynamicScale,  screens::{  gameplay::{arena::ArenaHeightOffset, networking::NetworkMapping},  GameState,  }, };  use super::{AssociatedTower, IntoTag, SpawnDirection};  pub(super) fn plugin(app: &mut App) {  app.add\_observer(spawn\_king\_tower);   app.configure\_loading\_state(  LoadingStateConfig::new(GameState::Loading).load\_collection::<KingTowerAssets>(),  );   app.add\_systems(OnExit(GameState::Gameplay), despawn\_king\_towers); }  #[derive(Event)] pub struct SpawnKingTower(pub Entity, pub ArenaPos, pub PlayerNumber);  #[derive(Component)] #[require(  DynamicScale(|| DynamicScale(0.75)),  ArenaHeightOffset(|| ArenaHeightOffset(1.3)), )] struct KingTower;  #[derive(Component)] #[require(  Health,   Name(|| Name::new("Король на башне")),  DynamicScale(|| DynamicScale(0.55)),  UnitState,  ArenaHeightOffset(|| ArenaHeightOffset(3.)), )] struct KingTowerKing;  #[derive(Resource, AssetCollection)] struct KingTowerAssets {  #[asset(path = "units/king\_tower/ally\_tower.aseprite")]  ally\_tower: Handle<Aseprite>,  #[asset(path = "units/king\_tower/enemy\_tower.aseprite")]  enemy\_tower: Handle<Aseprite>,   #[asset(path = "units/priest/ally\_priest.aseprite")]  ally\_king: Handle<Aseprite>,  #[asset(path = "units/priest/enemy\_priest.aseprite")]  enemy\_king: Handle<Aseprite>, }  fn spawn\_king\_tower(  trigger: Trigger<SpawnKingTower>,  mut cmd: Commands,  self\_num: Res<PlayerNumber>,  assets: ResMut<KingTowerAssets>,  mut network\_mapping: ResMut<NetworkMapping>, ) {  let SpawnKingTower(entity, mut pos, player\_num) = trigger.event();   let direction = self\_num.spawn\_direction(\*player\_num);   let (tower\_sprite, king\_sprite) = if pos.1 < 0. {  (assets.ally\_tower.clone(), assets.ally\_king.clone())  } else {  (assets.enemy\_tower.clone(), assets.enemy\_king.clone())  };   pos.1 += 0.01;  let tower = cmd  .spawn((  KingTower,  pos,  AseSpriteSlice {  name: "king\_tower".into(),  aseprite: tower\_sprite,  },  ))  .id();   pos.1 -= 0.01;  let king = cmd  .spawn((  KingTowerKing,  pos,  direction,  AseSpriteAnimation {  animation: Animation::tag(direction.tag()),  aseprite: king\_sprite,  },  AssociatedTower(tower),  ))  .id();  network\_mapping.insert(\*entity, king); }  fn despawn\_king\_towers(mut cmd: Commands, towers: Query<(Entity, &AssociatedTower)>) {  for (king, tower) in towers.iter() {  cmd.entity(tower.0).despawn();  cmd.entity(king).despawn();  } } |

|  |
| --- |
| boyar\_tournament\src\screens\gameplay\units\mod.rs |
| use archer\_tower::SpawnArcherTower; use bat::SpawnBat; use bevy::prelude::\*; use bevy\_aseprite\_ultra::prelude::{AnimationState, AseSpriteAnimation, Aseprite}; use bomber::SpawnBomber; use common::{ArenaPos, Direction, Health, PlayerNumber, Unit, UnitState}; use king\_tower::SpawnKingTower; use musketeer::SpawnMusketeer; use priest::SpawnPriest; use rus::SpawnRus; use giant::SpawnGiant;  use crate::screens::GameState;  mod archer\_tower; mod bat; mod bomber; mod king\_tower; mod musketeer; mod priest; mod rus; mod giant;  pub(super) fn plugin(app: &mut App) {  app.register\_type::<Direction>();  app.register\_type::<UnitState>();  app.register\_type::<Health>();   app.add\_systems(  PreUpdate,  manage\_animation.run\_if(in\_state(GameState::Gameplay)),  );   app.add\_plugins((  archer\_tower::plugin,  king\_tower::plugin,  rus::plugin,  musketeer::plugin,  bat::plugin,  priest::plugin,  bomber::plugin,  giant::plugin,  )); }  fn manage\_animation(  mut animation\_query: Query<(  &Direction,  &UnitState,  &mut AseSpriteAnimation,  &mut AnimationState,  )>,  aseprites: Res<Assets<Aseprite>>, ) {  for (direction, state, mut animation, mut animation\_state) in animation\_query.iter\_mut() {  match state {  UnitState::Idle => {  let tag\_meta = aseprites  .get(animation.aseprite.id())  .unwrap()  .tags  .get(direction.tag())  .unwrap();  let start\_frame = tag\_meta.range.start();  animation\_state.current\_frame = \*start\_frame;   animation.animation.tag = Some(direction.tag().into());  }  UnitState::Moving => {  let tag\_meta = aseprites  .get(animation.aseprite.id())  .unwrap()  .tags  .get(direction.tag())  .unwrap();  let start\_frame = tag\_meta.range.start();  let end\_frame = tag\_meta.range.end();  if animation\_state.current\_frame < \*start\_frame  || animation\_state.current\_frame > \*end\_frame  {  animation\_state.current\_frame = \*start\_frame;  }   animation.animation.tag = Some(direction.tag().into());  }  UnitState::Attacking => {  let mut tag = String::from(direction.tag());  tag.push('a');   let tag\_meta = aseprites  .get(animation.aseprite.id())  .unwrap()  .tags  .get(&tag)  .unwrap();  let start\_frame = tag\_meta.range.start();  let end\_frame = tag\_meta.range.end();  if animation\_state.current\_frame < \*start\_frame  || animation\_state.current\_frame > \*end\_frame  {  animation\_state.current\_frame = \*start\_frame;  }   animation.animation.tag = Some(tag);  }  }  } }  /// Требуется для привязки юнита к башне #[derive(Component)] pub struct AssociatedTower(pub Entity);  pub(super) trait SpawnUnit {  fn spawn(  &self,  entity: Entity,  pos: ArenaPos,  player\_num: PlayerNumber,  cmd: &mut Commands,  ); }  impl SpawnUnit for Unit {  fn spawn(  &self,  entity: Entity,  pos: ArenaPos,  player\_num: PlayerNumber,  cmd: &mut Commands,  ) {  match self {  Unit::ArcherTower => cmd.trigger(SpawnArcherTower(entity, pos, player\_num)),  Unit::KingTower => cmd.trigger(SpawnKingTower(entity, pos, player\_num)),  Unit::Rus => cmd.trigger(SpawnRus(entity, pos, player\_num)),  Unit::Musketeer => cmd.trigger(SpawnMusketeer(entity, pos, player\_num)),  Unit::Bat => cmd.trigger(SpawnBat(entity, pos, player\_num)),  Unit::Priest => cmd.trigger(SpawnPriest(entity, pos, player\_num)),  Unit::Bomber => cmd.trigger(SpawnBomber(entity, pos, player\_num)),  Unit::Giant => cmd.trigger(SpawnGiant(entity, pos, player\_num)),  }  } }  trait SpawnDirection {  fn spawn\_direction(self, player\_num: Self) -> Direction; } impl SpawnDirection for PlayerNumber {  fn spawn\_direction(self, player\_num: PlayerNumber) -> Direction {  use PlayerNumber::\*;  match (self, player\_num) {  (One, One) | (Two, Two) => Direction::Up,  \_ => Direction::Down,  }  } }  trait IntoTag {  fn tag(&self) -> &'static str; } impl IntoTag for Direction {  fn tag(&self) -> &'static str {  match self {  Direction::Up => "u",  Direction::Down => "d",  Direction::Left => "l",  Direction::Right => "r",  }  } } impl IntoTag for UnitState {  fn tag(&self) -> &'static str {  match self {  UnitState::Idle => "",  UnitState::Moving => "",  UnitState::Attacking => "a",  }  } } |

|  |
| --- |
| boyar\_tournament\src\screens\gameplay\units\musketeer.rs |
| use bevy::prelude::\*; use bevy\_aseprite\_ultra::prelude::\*; use bevy\_asset\_loader::prelude::\*; use common::{ArenaPos, Health, PlayerNumber, UnitState};  use crate::{  scaling::DynamicScale,  screens::{  gameplay::{arena::ArenaHeightOffset, networking::NetworkMapping},  GameState,  }, };  use super::{IntoTag, SpawnDirection};  pub(super) fn plugin(app: &mut App) {  app.add\_observer(spawn\_musketeer);   app.configure\_loading\_state(  LoadingStateConfig::new(GameState::Loading).load\_collection::<MusketeerAssets>(),  ); }  #[derive(Event)] pub struct SpawnMusketeer(pub Entity, pub ArenaPos, pub PlayerNumber);  #[derive(Component)] #[require(  Health,   Name(|| Name::new("Стрелок")),  DynamicScale(|| DynamicScale(0.55)),  UnitState,  ArenaHeightOffset(|| ArenaHeightOffset(1.3)), )] struct Musketeer;  #[derive(Resource, AssetCollection)] struct MusketeerAssets {  #[asset(path = "units/musketeer/musketeer.aseprite")]  sprite: Handle<Aseprite>, }  fn spawn\_musketeer(  trigger: Trigger<SpawnMusketeer>,  mut cmd: Commands,  self\_num: Res<PlayerNumber>,  assets: ResMut<MusketeerAssets>,  mut network\_mapping: ResMut<NetworkMapping>, ) {  let &SpawnMusketeer(entity, pos, player\_num) = trigger.event();   let direction = self\_num.spawn\_direction(player\_num);   let musketeer = cmd  .spawn((  Musketeer,  pos,  direction,  AseSpriteAnimation {  animation: Animation::tag(direction.tag()),  aseprite: assets.sprite.clone(),  },  ))  .id();   network\_mapping.insert(entity, musketeer); } |

|  |
| --- |
| boyar\_tournament\src\screens\gameplay\units\priest.rs |
| use bevy::prelude::\*; use bevy\_aseprite\_ultra::prelude::\*; use bevy\_asset\_loader::prelude::\*; use common::{ArenaPos, Health, PlayerNumber, UnitState};  use crate::{  scaling::DynamicScale,  screens::{  gameplay::{arena::ArenaHeightOffset, networking::NetworkMapping},  GameState,  }, };  use super::{IntoTag, SpawnDirection};  pub(super) fn plugin(app: &mut App) {  app.add\_observer(spawn\_priest);   app.configure\_loading\_state(  LoadingStateConfig::new(GameState::Loading).load\_collection::<PriestAssets>(),  ); }  #[derive(Event)] pub struct SpawnPriest(pub Entity, pub ArenaPos, pub PlayerNumber);  #[derive(Component)] #[require(  Health,   Name(|| Name::new("Жрец")),  DynamicScale(|| DynamicScale(0.55)),  UnitState,  ArenaHeightOffset(|| ArenaHeightOffset(1.1)), )] struct Priest;  #[derive(Resource, AssetCollection)] struct PriestAssets {  #[asset(path = "units/priest/priest.aseprite")]  sprite: Handle<Aseprite>, }  fn spawn\_priest(  trigger: Trigger<SpawnPriest>,  mut cmd: Commands,  self\_num: Res<PlayerNumber>,  assets: ResMut<PriestAssets>,  mut network\_mapping: ResMut<NetworkMapping>, ) {  let &SpawnPriest(entity, pos, player\_num) = trigger.event();   let direction = self\_num.spawn\_direction(player\_num);   let priest = cmd  .spawn((  Priest,  pos,  direction,  AseSpriteAnimation {  animation: Animation::tag(direction.tag()),  aseprite: assets.sprite.clone(),  },  ))  .id();   network\_mapping.insert(entity, priest); } |

|  |
| --- |
| boyar\_tournament\src\screens\gameplay\units\rus.rs |
| use bevy::prelude::\*; use bevy\_aseprite\_ultra::prelude::\*; use bevy\_asset\_loader::prelude::\*; use common::{ArenaPos, Health, PlayerNumber, UnitState};  use crate::{  scaling::DynamicScale,  screens::{  gameplay::{arena::ArenaHeightOffset, networking::NetworkMapping},  GameState,  }, };  use super::{IntoTag, SpawnDirection};  pub(super) fn plugin(app: &mut App) {  app.add\_observer(spawn\_rus);   app.configure\_loading\_state(  LoadingStateConfig::new(GameState::Loading).load\_collection::<RusAssets>(),  ); }  #[derive(Event)] pub struct SpawnRus(pub Entity, pub ArenaPos, pub PlayerNumber);  #[derive(Component)] #[require(  Health,   Name(|| Name::new("Рус")),  DynamicScale(|| DynamicScale(0.6)),  UnitState,  ArenaHeightOffset(|| ArenaHeightOffset(2.)), )] struct Rus;  #[derive(Resource, AssetCollection)] struct RusAssets {  #[asset(path = "units/rus/rus.aseprite")]  sprite: Handle<Aseprite>, }  fn spawn\_rus(  trigger: Trigger<SpawnRus>,  mut cmd: Commands,  self\_num: Res<PlayerNumber>,  assets: ResMut<RusAssets>,  mut network\_mapping: ResMut<NetworkMapping>, ) {  let &SpawnRus(entity, pos, player\_num) = trigger.event();   let direction = self\_num.spawn\_direction(player\_num);   let rus = cmd  .spawn((  Rus,  pos,  direction,  AseSpriteAnimation {  animation: Animation::tag(direction.tag()),  aseprite: assets.sprite.clone(),  },  ))  .id();   network\_mapping.insert(entity, rus); } |

|  |
| --- |
| common\src\lib.rs |
| use std::{  net::Ipv4Addr,  ops::{AddAssign, Sub, SubAssign}, };  use bevy::{math::vec2, prelude::\*}; use bevy\_quinnet::shared::channels::{ChannelId, ChannelType, ChannelsConfiguration}; use serde::{Deserialize, Serialize};  //pub const SERVER\_HOST: Ipv4Addr = Ipv4Addr::new(178, 71, 57, 127); pub const SERVER\_HOST: Ipv4Addr = Ipv4Addr::LOCALHOST; pub const LOCAL\_BIND\_IP: Ipv4Addr = Ipv4Addr::UNSPECIFIED; pub const SERVER\_PORT: u16 = 50505;  #[derive(  Debug,  Component,  Reflect,  Serialize,  Deserialize,  Clone,  Copy,  Default,  PartialEq,  PartialOrd, )] #[reflect(Component)] pub struct ArenaPos(pub f32, pub f32); impl Sub for ArenaPos {  type Output = Self;   fn sub(self, rhs: Self) -> Self::Output {  ArenaPos(self.0 - rhs.0, self.1 - rhs.1)  } } impl SubAssign for ArenaPos {  fn sub\_assign(&mut self, rhs: Self) {  self.0 -= rhs.0;  self.1 -= rhs.1;  } } impl AddAssign for ArenaPos {  fn add\_assign(&mut self, rhs: Self) {  self.0 += rhs.0;  self.1 += rhs.1;  } } impl ArenaPos {  pub fn normalize(&self) -> Self {  let v = vec2(self.0, self.1).normalize();  ArenaPos(v.x, v.y)  }  pub fn mul(&self, n: f32) -> Self {  ArenaPos(self.0 \* n, self.1 \* n)  }  pub fn distance(&self, rhs: &Self) -> f32 {  ((self.0 - rhs.0).powi(2) + (self.1 - rhs.1).powi(2)).sqrt()  }  pub fn direction(&self, rhs: &Self) -> Self {  if self.distance(rhs) < 0.01 {  return ArenaPos(0., 0.);  }  (\*rhs - \*self).normalize()  } }  #[derive(Debug, Component, Serialize, Deserialize, Clone, Copy, Reflect)] #[reflect(Component)] pub enum Card {  Rus,  Musketeer,  ThreeMusketeers,  Priest,  Bats,  BatHorde,  Bomber,  Giant, }  #[derive(Debug, Component, Serialize, Deserialize, Clone, Copy)] pub enum Unit {  ArcherTower,  KingTower,  Rus,  Musketeer,  Bat,  Priest,  Bomber,  Giant, }  #[derive(Debug, Component, Serialize, Deserialize, Clone, Copy)] pub enum Projectile {  Bullet,  Fireball,  Bomb, }  #[derive(Component, Reflect, Serialize, Deserialize, Clone, Copy)] #[reflect(Component)] pub struct Health(pub u16, pub u16); // Текущее и максимальное здоровье impl Health {  // Конкретное значение указывается в сервере, default для спауна на клиенте  pub fn new(amount: u16) -> Self {  Health(amount, amount)  } } impl Default for Health {  fn default() -> Self {  Self::new(100)  } }  #[derive(Component, Debug, Serialize, Deserialize, Clone, Copy, Reflect, Default)] #[reflect(Component)] pub enum Direction {  #[default]  Up,  Down,  Left,  Right, } impl Direction {  pub fn opposite(&self) -> Self {  use Direction::\*;  match self {  Up => Down,  Down => Up,  Left => Right,  Right => Left,  }  } }  #[derive(Component, Debug, Serialize, Deserialize, Clone, Copy, Reflect, Default)] #[reflect(Component)] pub enum UnitState {  #[default]  Idle, // Для построек, а также отправляется клиенту для юнитов в стане  Moving, // Для всего остального  Attacking, }  #[derive(Serialize, Deserialize)] pub enum ClientMessage {  PlayCard { card: Card, placement: ArenaPos }, }  #[derive(  Resource,  Component,  Serialize,  Deserialize,  Debug,  Clone,  Copy,  Hash,  Eq,  PartialEq,  Default, )] pub enum PlayerNumber {  #[default]  One, // Игрок "снизу"  Two, // Игрок "сверху" }  #[derive(Serialize, Deserialize)] pub enum ServerMessage {  StartGame(PlayerNumber),  SpawnUnit {  server\_entity: Entity,  unit: Unit,  pos: ArenaPos,  owner: PlayerNumber,  },  SpawnProjectile {  server\_entity: Entity,  projectile: Projectile,  attacker: Entity,  receiver: Entity,  pos: ArenaPos,  },  Despawn(Entity),  SyncEntities {  units: Vec<(Entity, ArenaPos, Direction, UnitState, Health)>,  projectiles: Vec<(Entity, ArenaPos)>,  }, }  #[repr(u8)] pub enum ClientChannel {  // Разыгрывание карт, и мб вызов эмоутов  OrderedReliable, } impl From<ClientChannel> for ChannelId {  fn from(value: ClientChannel) -> Self {  value as \_  } } impl ClientChannel {  pub fn channels\_config() -> ChannelsConfiguration {  ChannelsConfiguration::from\_types(vec![ChannelType::OrderedReliable]).unwrap()  } }  #[repr(u8)] pub enum ServerChannel {  // Инициализация  OrderedReliable,  // Рассылка действий игроков  UnorderedReliable,  // Синхронизация юнитов  Unreliable, } impl From<ServerChannel> for ChannelId {  fn from(value: ServerChannel) -> Self {  value as \_  } } impl ServerChannel {  pub fn channels\_config() -> ChannelsConfiguration {  ChannelsConfiguration::from\_types(vec![  ChannelType::OrderedReliable,  ChannelType::UnorderedReliable,  ChannelType::Unreliable,  ])  .unwrap()  } } |

|  |
| --- |
| desktop\_client\src\main.rs |
| // Выключить коммандную строку в релиз-сборках для Windows #![cfg\_attr(not(debug\_assertions), windows\_subsystem = "windows")]  use bevy::{asset::AssetMetaCheck, prelude::\*, window::PrimaryWindow, winit::WinitWindows}; use boyar\_tournament::GamePlugin; use std::io::Cursor; use winit::window::Icon;  fn main() {  App::new()  .add\_plugins(  DefaultPlugins  .set(WindowPlugin {  primary\_window: Window {  title: "Боярский Турнир".into(),  ..default()  }  .into(),  ..default()  })  .set(ImagePlugin::default\_nearest())  .set(AssetPlugin {  meta\_check: AssetMetaCheck::Never,  file\_path: "../assets".into(),  processed\_file\_path: "../assets".into(),  ..default()  }),  )  .add\_plugins(GamePlugin)  .add\_systems(Startup, set\_window\_icon)  .run(); }  fn set\_window\_icon(  windows: NonSend<WinitWindows>,  primary\_window: Query<Entity, With<PrimaryWindow>>, ) {  let primary\_window = primary\_window.single();  let Some(primary\_window) = windows.get\_window(primary\_window) else {  return;  };  let icon\_buf = Cursor::new(include\_bytes!("../../assets/icons/desktop\_icon.png"));  if let Ok(image) = image::load(icon\_buf, image::ImageFormat::Png) {  let image = image.into\_rgba8();  let (width, height) = image.dimensions();  let rgba = image.into\_raw();  let icon = Icon::from\_rgba(rgba, width, height).unwrap();  primary\_window.set\_window\_icon(Some(icon));  } } |

|  |
| --- |
| mobile\_client\src\lib.rs |
| use bevy::prelude::\*; use bevy::window::{AppLifecycle, WindowMode}; use bevy::winit::WinitSettings; use boyar\_tournament::GamePlugin;  #[bevy\_main] fn main() {  App::new()  .add\_plugins((  DefaultPlugins  .set(WindowPlugin {  primary\_window: Window {  resizable: false,  mode: WindowMode::BorderlessFullscreen(MonitorSelection::Primary),  ..default()  }  .into(),  ..default()  })  .set(ImagePlugin::default\_nearest()),  GamePlugin,  ))  .insert\_resource(WinitSettings::mobile())  .add\_systems(Update, handle\_lifetime)  .run(); }  /// Остановка звука при переходе приложения в фоновый режим // Взято из официального android примера, не проверял нужно или нет fn handle\_lifetime(  mut lifecycle\_events: EventReader<AppLifecycle>,  music\_controller: Single<&AudioSink>, ) {  for event in lifecycle\_events.read() {  match event {  AppLifecycle::Idle | AppLifecycle::WillSuspend | AppLifecycle::WillResume => {}  AppLifecycle::Suspended => music\_controller.pause(),  AppLifecycle::Running => music\_controller.play(),  }  } } |

|  |
| --- |
| server\src\ai.rs |
| use bevy::prelude::\*; use bevy\_quinnet::server::QuinnetServer; use common::{  ArenaPos, Health, PlayerNumber, Projectile, ServerChannel, ServerMessage, UnitState, };  use crate::{projectiles::SpawnProjectile, units::UnitType};  pub(super) fn plugin(app: &mut App) {  app.add\_systems(  FixedUpdate,  (  (  update\_attacks,  update\_unit\_state,  update\_stun\_timers,  update\_movement,  ),  check\_health,  )  .chain(),  ); }  pub enum AttackType {  Melee(u16), // Урон  Ranged(Projectile), }  pub enum AttackTargetType {  Ground,  All, }  #[derive(Component)] pub struct Attack {  pub target: Option<Entity>,  pub a\_type: AttackType,  pub t\_type: AttackTargetType,  pub cooldown\_timer: Timer,  pub range: f32, } impl Attack {  pub fn new(a\_type: AttackType, targets: AttackTargetType, cd: f32, range: f32) -> Self {  Self {  target: None,  a\_type,  t\_type: targets,  cooldown\_timer: Timer::from\_seconds(cd, TimerMode::Repeating),  range,  }  } }  fn update\_attacks(  mut attacks: Query<(Entity, &mut Attack)>,  mut units: Query<(&ArenaPos, &mut Health)>,  time: Res<Time>,  mut cmd: Commands, ) {  for (attacker, mut attack) in &mut attacks {  // target есть только в UnitState::Attacking  let Some(receiver) = attack.target else {  attack.cooldown\_timer.reset();  continue;  };  let Ok((\_, mut health)) = units.get\_mut(receiver) else {  // Все мертвы  attack.target = None;  continue;  };  if !attack.cooldown\_timer.tick(time.delta()).just\_finished() {  continue;  }   match attack.a\_type {  AttackType::Melee(damage) => health.0 = health.0.saturating\_sub(damage),  AttackType::Ranged(projectile) => {  let (pos, \_) = units.get(attacker).unwrap();  projectile.spawn(attacker, receiver, \*pos, &mut cmd)  }  }  } }  fn check\_health(  query: Query<(Entity, &Health)>,  mut server: ResMut<QuinnetServer>,  mut cmd: Commands, ) {  for (entity, health) in &query {  if health.0 == 0 {  cmd.entity(entity).despawn();  server  .endpoint\_mut()  .broadcast\_message\_on(  ServerChannel::OrderedReliable,  ServerMessage::Despawn(entity),  )  .unwrap();  }  } }  #[derive(Component)] pub struct Movement {  pub target: Option<Entity>,  pub speed: f32, } impl Movement {  pub fn new(speed: f32) -> Self {  Self {  target: None,  speed,  }  } } fn update\_movement(  mut query: Query<(Entity, &mut Movement), Without<StunnedTimer>>,  states: Query<&UnitState>,  mut positions: Query<&mut ArenaPos>,  time: Res<Time>, ) {  for (entity, mut movement) in &mut query {  if let Ok(state) = states.get(entity) {  let UnitState::Moving = state else {  continue;  };  }  // target устанавливается в update\_unit\_state  let Some(target) = movement.target else {  continue;  };  let Ok(&target\_pos) = positions.get(target) else {  movement.target = None;  continue;  };  let Ok(mut self\_pos) = positions.get\_mut(entity) else {  continue;  };  let direction = self\_pos.direction(&target\_pos);  \*self\_pos += direction.mul(movement.speed \* time.delta\_secs());  } }  #[derive(Component)] // Таймер добавляется при спавне юнитов pub struct StunnedTimer(pub Timer); impl Default for StunnedTimer {  fn default() -> Self {  Self(Timer::from\_seconds(1.5, TimerMode::Once))  } } fn update\_stun\_timers(  mut query: Query<(Entity, &mut StunnedTimer)>,  mut cmd: Commands,  time: Res<Time>, ) {  for (entity, mut timer) in &mut query {  if timer.0.tick(time.delta()).just\_finished() {  cmd.entity(entity).remove::<StunnedTimer>();  }  } }  #[derive(Component)] pub struct AggroRadius(pub f32);  fn update\_unit\_state(  mut attackers: Query<  (  Entity,  &mut UnitState,  &mut Attack,  Option<&AggroRadius>,  Option<&mut Movement>,  ),  Without<StunnedTimer>,  >,  receivers: Query<(Entity, &ArenaPos, &PlayerNumber, &UnitType)>,  towers: Query<(Entity, &ArenaPos, &PlayerNumber), Without<Movement>>, ) {  'outer: for (self\_entity, mut state, mut attack, aggro\_radius, mut movement) in  &mut attackers  {  match \*state {  UnitState::Idle | UnitState::Moving => {  let (\_, self\_pos, self\_player\_numer, \_) = receivers.get(self\_entity).unwrap();   for (entity, pos, player\_number, unit\_type) in &receivers {  if self\_player\_numer == player\_number {  // Своих не бьём  continue;  }  if let (AttackTargetType::Ground, UnitType::Air) =  (&attack.t\_type, unit\_type)  {  continue;  }   if self\_pos.distance(pos) <= attack.range {  \*state = UnitState::Attacking;  attack.target = Some(entity);  continue 'outer;  }   // У всего что не является постройкой есть и AggroRadius и Movement  if let (Some(aggro\_radius), Some(movement)) =  (aggro\_radius, movement.as\_mut())  {  if self\_pos.distance(pos) <= aggro\_radius.0 {  movement.target = Some(entity);  continue 'outer;  }  }  }   // Если никого нет вблизи, двигаемся к ближайшей башне  let Some(movement) = movement.as\_mut() else {  continue;  };  let mut nearest\_tower = None;  let mut minimal\_distance = 1000.;  for (tower\_entity, tower\_pos, tower\_player\_number) in &towers {  let distance = self\_pos.distance(tower\_pos);  if self\_player\_numer == tower\_player\_number || distance > minimal\_distance  {  continue;  }   minimal\_distance = distance;  nearest\_tower = Some(tower\_entity);  }  let Some(nearest\_tower) = nearest\_tower else {  // Башен врага не осталось, игра должна закончиться  continue;  };  movement.target = Some(nearest\_tower);  }  UnitState::Attacking => {  if let Some(target) = attack.target {  let (\_, self\_pos, \_, \_) = receivers.get(self\_entity).unwrap();   if let Ok((\_, pos, \_, \_)) = receivers.get(target) {  if self\_pos.distance(pos) > attack.range {  match movement.as\_mut() {  Some(\_) => \*state = UnitState::Moving,  None => \*state = UnitState::Idle,  }  }  continue;  };   continue;  }   match movement.as\_mut() {  Some(\_) => \*state = UnitState::Moving,  None => \*state = UnitState::Idle,  }  }  }  } } |

|  |
| --- |
| server\src\main.rs |
| use bevy::{log::LogPlugin, prelude::\*};  mod ai; mod networking; mod projectiles; mod units;  fn main() {  App::new()  .add\_plugins((  MinimalPlugins,  LogPlugin::default(),  ai::plugin,  units::plugin,  projectiles::plugin,  networking::plugin,  ))  .run(); } |

|  |
| --- |
| server\src\networking.rs |
| use core::f32;  use bevy::{prelude::\*, utils::HashMap}; use bevy\_quinnet::{  server::{  certificate::CertificateRetrievalMode, ConnectionEvent, QuinnetServer,  QuinnetServerPlugin, ServerEndpointConfiguration,  },  shared::ClientId, }; use common::{  ArenaPos, Card, ClientMessage, Direction, Health, PlayerNumber, ServerChannel,  ServerMessage, Unit, UnitState, LOCAL\_BIND\_IP, SERVER\_HOST, SERVER\_PORT, };  use crate::{  ai::{Attack, Movement, StunnedTimer},  units::{Giant, SpawnUnit}, };  pub(super) fn plugin(app: &mut App) {  app.add\_plugins(QuinnetServerPlugin::default());   app.init\_resource::<Lobby>();  app.add\_systems(Startup, start\_listening);  app.add\_systems(Update, (handle\_connection\_events, handle\_client\_messages));   app.add\_systems(FixedPostUpdate, sync\_entities); }  fn start\_listening(mut server: ResMut<QuinnetServer>) {  server  .start\_endpoint(  ServerEndpointConfiguration::from\_ip(LOCAL\_BIND\_IP, SERVER\_PORT),  CertificateRetrievalMode::GenerateSelfSigned {  server\_hostname: SERVER\_HOST.to\_string(),  },  ServerChannel::channels\_config(),  )  .unwrap(); }  #[derive(Resource, Default, Deref, DerefMut)] pub struct Lobby(HashMap<ClientId, PlayerNumber>);  fn handle\_connection\_events(  mut connection\_events: EventReader<ConnectionEvent>,  mut lobby: ResMut<Lobby>,  mut server: ResMut<QuinnetServer>,  mut cmd: Commands, ) {  let lobby\_len = lobby.len() as u8;  for client in connection\_events.read() {  if lobby\_len >= 2 {  server.endpoint\_mut().disconnect\_client(client.id).unwrap();  continue;  }  use PlayerNumber::\*;   let player\_num = match lobby\_len {  0 => One,  1 => Two,  \_ => unreachable!(),  };  lobby.insert(client.id, player\_num);   if lobby.len() == 2 {  // Отправить каждому игроку его PlayerNumber  for (client\_id, player\_num) in lobby.iter() {  server  .endpoint\_mut()  .send\_message\_on(  \*client\_id,  ServerChannel::OrderedReliable,  ServerMessage::StartGame(\*player\_num),  )  .unwrap();  }   Unit::ArcherTower.spawn(ArenaPos(-5.5, -9.5), One, &mut cmd);  Unit::KingTower.spawn(ArenaPos(0., -13.), One, &mut cmd);  Unit::ArcherTower.spawn(ArenaPos(5.5, -9.5), One, &mut cmd);   Unit::ArcherTower.spawn(ArenaPos(-5.5, 9.5), Two, &mut cmd);  Unit::KingTower.spawn(ArenaPos(0., 13.), Two, &mut cmd);  Unit::ArcherTower.spawn(ArenaPos(5.5, 9.5), Two, &mut cmd);  }  } }  fn handle\_client\_messages(  mut server: ResMut<QuinnetServer>,  lobby: Res<Lobby>,  mut cmd: Commands, ) {  let endpoint = server.endpoint\_mut();  for client\_id in endpoint.clients() {  while let Some((\_, message)) =  endpoint.try\_receive\_message\_from::<ClientMessage>(client\_id)  {  let player\_num = lobby.get(&client\_id).unwrap();  match message {  ClientMessage::PlayCard { card, placement } => match card {  Card::Rus => Unit::Rus.spawn(placement, \*player\_num, &mut cmd),  Card::Musketeer => Unit::Musketeer.spawn(placement, \*player\_num, &mut cmd),  Card::ThreeMusketeers => {  let ArenaPos(x, y) = placement;  Unit::Musketeer.spawn(ArenaPos(x, y + 0.8), \*player\_num, &mut cmd);  Unit::Musketeer.spawn(ArenaPos(x + 0.8, y), \*player\_num, &mut cmd);  Unit::Musketeer.spawn(ArenaPos(x - 0.8, y), \*player\_num, &mut cmd);  }  Card::Bats => {  let ArenaPos(x, y) = placement;  Unit::Bat.spawn(ArenaPos(x, y + 0.8), \*player\_num, &mut cmd);  Unit::Bat.spawn(ArenaPos(x + 0.8, y), \*player\_num, &mut cmd);  Unit::Bat.spawn(ArenaPos(x - 0.8, y), \*player\_num, &mut cmd);  }  Card::BatHorde => {  let ArenaPos(x, y) = placement;  Unit::Bat.spawn(ArenaPos(x + 0.5, y + 0.5), \*player\_num, &mut cmd);  Unit::Bat.spawn(ArenaPos(x + 0.8, y), \*player\_num, &mut cmd);  Unit::Bat.spawn(ArenaPos(x + 0.5, y - 0.5), \*player\_num, &mut cmd);  Unit::Bat.spawn(ArenaPos(x - 0.5, y - 0.5), \*player\_num, &mut cmd);  Unit::Bat.spawn(ArenaPos(x - 0.8, y), \*player\_num, &mut cmd);  Unit::Bat.spawn(ArenaPos(x - 0.5, y + 0.5), \*player\_num, &mut cmd);  }  Card::Priest => Unit::Priest.spawn(placement, \*player\_num, &mut cmd),  Card::Bomber => Unit::Bomber.spawn(placement, \*player\_num, &mut cmd),  Card::Giant => Unit::Giant.spawn(placement, \*player\_num, &mut cmd),  },  }  }  } }  trait DefaultDirection {  fn default\_direction(&self) -> Direction; } impl DefaultDirection for PlayerNumber {  fn default\_direction(&self) -> Direction {  match self {  PlayerNumber::One => Direction::Up,  PlayerNumber::Two => Direction::Down,  }  } }  fn calc\_direction(direction: &ArenaPos) -> Direction {  let mut angle = direction.0.acos() \* 180. / f32::consts::PI;  if direction.1 < 0. {  angle = -angle + 360.;  }   match angle {  0.0..20. | 340.0..360. => Direction::Right,  20.0..160. => Direction::Up,  160.0..200. => Direction::Left,  200.0..340. => Direction::Down,  \_ => Direction::Right,  } }  fn sync\_entities(  units: Query<(  Entity,  &ArenaPos,  &UnitState,  &Attack,  Option<&Movement>,  &PlayerNumber,  &Health,  Option<&StunnedTimer>,  )>,  giants: Query<(  Entity,  &ArenaPos,  &UnitState,  &Giant,  &Movement,  &PlayerNumber,  &Health,  Option<&StunnedTimer>,  )>,  projectiles: Query<(Entity, &ArenaPos), Without<PlayerNumber>>,  positions: Query<&ArenaPos>,  mut server: ResMut<QuinnetServer>, ) {  let mut u = Vec::new();  for (entity, pos, state, attack, movement, player\_num, health, stun) in &units {  let direction = match state {  UnitState::Idle => player\_num.default\_direction(),  UnitState::Moving => {  let movement = movement.unwrap();  match movement.target {  Some(m) => {  let Ok(target\_pos) = positions.get(m) else {  continue;  };  calc\_direction(&pos.direction(target\_pos))  }  None => player\_num.default\_direction(),  }  }  UnitState::Attacking => match attack.target {  Some(a) => {  let Ok(target\_pos) = positions.get(a) else {  continue;  };  calc\_direction(&pos.direction(target\_pos))  }  None => player\_num.default\_direction(),  },  };  let mut state = \*state;  if let Some(\_) = stun {  state = UnitState::Idle  }  u.push((entity, \*pos, direction, state, \*health));  }  for (entity, pos, state, giant, movement, player\_num, health, stun) in &giants {  let direction = match state {  UnitState::Idle => player\_num.default\_direction(),  UnitState::Moving => match movement.target {  Some(m) => {  let Ok(target\_pos) = positions.get(m) else {  continue;  };  calc\_direction(&pos.direction(target\_pos))  }  None => player\_num.default\_direction(),  },  UnitState::Attacking => match giant.target {  Some(a) => {  let Ok(target\_pos) = positions.get(a) else {  continue;  };  calc\_direction(&pos.direction(target\_pos))  }  None => player\_num.default\_direction(),  },  };  let mut state = \*state;  if let Some(\_) = stun {  state = UnitState::Idle  }  u.push((entity, \*pos, direction, state, \*health));  }   let mut p = Vec::new();  for (entity, position) in &projectiles {  p.push((entity, \*position));  }   server  .endpoint\_mut()  .broadcast\_message\_on(  ServerChannel::Unreliable,  ServerMessage::SyncEntities {  units: u,  projectiles: p,  },  )  .unwrap(); } |

|  |
| --- |
| server\src\projectiles\bomb.rs |
| use bevy::prelude::\*; use bevy\_quinnet::server::QuinnetServer; use common::{ArenaPos, Health, PlayerNumber, Projectile, ServerChannel, ServerMessage};  use crate::{ai::Movement, units::{Hitbox, UnitType}};  use super::ProjectileRadius;  pub(super) fn plugin(app: &mut App) {  app.add\_observer(spawn\_bomb);   app.add\_systems(FixedUpdate, update\_bombs); }  #[derive(Event)] pub struct SpawnBomb(pub Entity, pub Entity, pub ArenaPos);  #[derive(Component)] #[require(  Projectile(|| Projectile::Bomb),  ProjectileRadius(|| ProjectileRadius(1.)), )] struct Bomb(Entity);  fn spawn\_bomb(  trigger: Trigger<SpawnBomb>,  mut server: ResMut<QuinnetServer>,  mut cmd: Commands, ) {  let &SpawnBomb(attacker, receiver, pos) = trigger.event();   let entity = cmd  .spawn((  Bomb(receiver),  pos,  Movement {  target: Some(receiver),  speed: 15.,  },  ))  .id();   server  .endpoint\_mut()  .broadcast\_message\_on(  ServerChannel::OrderedReliable,  ServerMessage::SpawnProjectile {  server\_entity: entity,  projectile: Projectile::Bomb,  attacker,  receiver,  pos,  },  )  .unwrap(); }  fn update\_bombs(  mut bombs: Query<  (Entity, &Bomb, &ProjectileRadius, &mut ArenaPos),  Without<PlayerNumber>,  >,  mut units: Query<(&ArenaPos, &mut Health, &Hitbox, &UnitType), With<PlayerNumber>>,  mut cmd: Commands,  mut server: ResMut<QuinnetServer>, ) {  for (entity, bomb, radius, pos) in &mut bombs {  let Ok((recv\_pos, \_, hitbox, \_)) = units.get\_mut(bomb.0) else {  // Цель умерла  cmd.entity(entity).despawn();  server  .endpoint\_mut()  .broadcast\_message\_on(  ServerChannel::OrderedReliable,  ServerMessage::Despawn(entity),  )  .unwrap();  continue;  };   if pos.distance(recv\_pos) > radius.0 + hitbox.0 {  continue;  }   for (recv\_pos, mut recv\_health, hitbox, unit\_type) in &mut units {  if let UnitType::Air = unit\_type {  continue;  }  if pos.distance(recv\_pos) > radius.0 + hitbox.0 {  continue;  }  recv\_health.0 = recv\_health.0.saturating\_sub(88);  }  cmd.entity(entity).despawn();  server  .endpoint\_mut()  .broadcast\_message\_on(  ServerChannel::OrderedReliable,  ServerMessage::Despawn(entity),  )  .unwrap();  } } |

|  |
| --- |
| server\src\projectiles\bullet.rs |
| use bevy::prelude::\*; use bevy\_quinnet::server::QuinnetServer; use common::{ArenaPos, Health, PlayerNumber, Projectile, ServerChannel, ServerMessage};  use crate::{ai::Movement, units::Hitbox};  use super::ProjectileRadius;  pub(super) fn plugin(app: &mut App) {  app.add\_observer(spawn\_bullet);   app.add\_systems(FixedUpdate, update\_bullets); }  #[derive(Event)] pub struct SpawnBullet(pub Entity, pub Entity, pub ArenaPos);  #[derive(Component)] #[require(  Projectile(|| Projectile::Bullet),  ProjectileRadius(|| ProjectileRadius(0.2)), )] struct Bullet(Entity);  fn spawn\_bullet(  trigger: Trigger<SpawnBullet>,  mut server: ResMut<QuinnetServer>,  mut cmd: Commands, ) {  let &SpawnBullet(attacker, receiver, pos) = trigger.event();   let entity = cmd  .spawn((  Bullet(receiver),  pos,  Movement {  target: Some(receiver),  speed: 40.,  },  ))  .id();   server  .endpoint\_mut()  .broadcast\_message\_on(  ServerChannel::OrderedReliable,  ServerMessage::SpawnProjectile {  server\_entity: entity,  projectile: Projectile::Bullet,  attacker,  receiver,  pos,  },  )  .unwrap(); }  fn update\_bullets(  mut bullets: Query<  (Entity, &Bullet, &ProjectileRadius, &mut ArenaPos),  Without<PlayerNumber>,  >,  mut units: Query<(&ArenaPos, &mut Health, &Hitbox), With<PlayerNumber>>,  mut cmd: Commands,  mut server: ResMut<QuinnetServer>, ) {  for (entity, bullet, radius, pos) in &mut bullets {  let Ok((recv\_pos, mut recv\_health, hitbox)) = units.get\_mut(bullet.0) else {  // Цель умерла  cmd.entity(entity).despawn();  server  .endpoint\_mut()  .broadcast\_message\_on(  ServerChannel::OrderedReliable,  ServerMessage::Despawn(entity),  )  .unwrap();  continue;  };   if pos.distance(recv\_pos) > radius.0 + hitbox.0 {  continue;  }   recv\_health.0 = recv\_health.0.saturating\_sub(50);  cmd.entity(entity).despawn();  server  .endpoint\_mut()  .broadcast\_message\_on(  ServerChannel::OrderedReliable,  ServerMessage::Despawn(entity),  )  .unwrap();  } } |

|  |
| --- |
| server\src\projectiles\fireball.rs |
| use bevy::prelude::\*; use bevy\_quinnet::server::QuinnetServer; use common::{ArenaPos, Health, PlayerNumber, Projectile, ServerChannel, ServerMessage};  use crate::{ai::Movement, units::Hitbox};  use super::ProjectileRadius;  pub(super) fn plugin(app: &mut App) {  app.add\_observer(spawn\_fireball);   app.add\_systems(FixedUpdate, update\_fireballs); }  #[derive(Event)] pub struct SpawnFireball(pub Entity, pub Entity, pub ArenaPos);  #[derive(Component)] #[require(  Projectile(|| Projectile::Fireball),  ProjectileRadius(|| ProjectileRadius(1.)), )] struct Fireball(Entity);  fn spawn\_fireball(  trigger: Trigger<SpawnFireball>,  mut server: ResMut<QuinnetServer>,  mut cmd: Commands, ) {  let &SpawnFireball(attacker, receiver, pos) = trigger.event();   let entity = cmd  .spawn((  Fireball(receiver),  pos,  Movement {  target: Some(receiver),  speed: 10.,  },  ))  .id();   server  .endpoint\_mut()  .broadcast\_message\_on(  ServerChannel::OrderedReliable,  ServerMessage::SpawnProjectile {  server\_entity: entity,  projectile: Projectile::Fireball,  attacker,  receiver,  pos,  },  )  .unwrap(); }  fn update\_fireballs(  mut fireballs: Query<  (Entity, &Fireball, &ProjectileRadius, &mut ArenaPos),  Without<PlayerNumber>,  >,  mut units: Query<(&ArenaPos, &mut Health, &Hitbox), With<PlayerNumber>>,  mut cmd: Commands,  mut server: ResMut<QuinnetServer>, ) {  for (entity, fireball, radius, pos) in &mut fireballs {  let Ok((recv\_pos, \_, hitbox)) = units.get\_mut(fireball.0) else {  // Цель умерла  cmd.entity(entity).despawn();  server  .endpoint\_mut()  .broadcast\_message\_on(  ServerChannel::OrderedReliable,  ServerMessage::Despawn(entity),  )  .unwrap();  continue;  };   if pos.distance(recv\_pos) > radius.0 + hitbox.0 {  continue;  }   for (recv\_pos, mut recv\_health, hitbox) in &mut units {  if pos.distance(recv\_pos) > radius.0 + hitbox.0 {  continue;  }  recv\_health.0 = recv\_health.0.saturating\_sub(140);  }  cmd.entity(entity).despawn();  server  .endpoint\_mut()  .broadcast\_message\_on(  ServerChannel::OrderedReliable,  ServerMessage::Despawn(entity),  )  .unwrap();  } } |

|  |
| --- |
| server\src\projectiles\mod.rs |
| use bevy::prelude::\*; use bomb::SpawnBomb; use bullet::SpawnBullet; use common::{ArenaPos, Projectile}; use fireball::SpawnFireball;  mod bomb; mod bullet; mod fireball;  pub(super) fn plugin(app: &mut App) {  app.add\_plugins((bullet::plugin, fireball::plugin, bomb::plugin)); }  #[derive(Component)] struct ProjectileRadius(pub f32);  pub(super) trait SpawnProjectile {  fn spawn(&self, attacker: Entity, receiver: Entity, pos: ArenaPos, cmd: &mut Commands); }  impl SpawnProjectile for Projectile {  fn spawn(&self, attacker: Entity, receiver: Entity, pos: ArenaPos, cmd: &mut Commands) {  match self {  Projectile::Bullet => cmd.trigger(SpawnBullet(attacker, receiver, pos)),  Projectile::Fireball => cmd.trigger(SpawnFireball(attacker, receiver, pos)),  Projectile::Bomb => cmd.trigger(SpawnBomb(attacker, receiver, pos)),  }  } } |

|  |
| --- |
| server\src\units\archer\_tower.rs |
| use bevy::prelude::\*; use bevy\_quinnet::server::QuinnetServer; use common::{  ArenaPos, Health, PlayerNumber, Projectile, ServerChannel, ServerMessage, Unit, UnitState, };  use crate::ai::{Attack, AttackTargetType, AttackType};  use super::{Hitbox, UnitType};  pub(super) fn plugin(app: &mut App) {  app.add\_observer(spawn\_archer\_tower); }  #[derive(Event)] pub struct SpawnArcherTower(pub ArenaPos, pub PlayerNumber);  #[derive(Component)] #[require(  Health(|| Health::new(1400)),  UnitType(|| UnitType::Ground),  UnitState,  Attack(|| Attack::new(AttackType::Ranged(Projectile::Bullet),  AttackTargetType::All, 0.75, 8.5)),  Hitbox(|| Hitbox(1.5)), )] struct ArcherTower;  fn spawn\_archer\_tower(  trigger: Trigger<SpawnArcherTower>,  mut server: ResMut<QuinnetServer>,  mut cmd: Commands, ) {  let &SpawnArcherTower(pos, owner) = trigger.event();   let entity = cmd.spawn((ArcherTower, pos, owner)).id();   server  .endpoint\_mut()  .broadcast\_message\_on(  ServerChannel::OrderedReliable,  ServerMessage::SpawnUnit {  server\_entity: entity,  unit: Unit::ArcherTower,  pos,  owner,  },  )  .unwrap(); } |

|  |
| --- |
| server\src\units\bat.rs |
| use bevy::prelude::\*; use bevy\_quinnet::server::QuinnetServer; use common::{ArenaPos, Health, PlayerNumber, ServerChannel, ServerMessage, Unit, UnitState};  use crate::ai::{AggroRadius, Attack, AttackTargetType, AttackType, Movement, StunnedTimer};  use super::{Hitbox, UnitType};  pub(super) fn plugin(app: &mut App) {  app.add\_observer(spawn\_bat); }  #[derive(Event)] pub struct SpawnBat(pub ArenaPos, pub PlayerNumber);  #[derive(Component)] #[require(  Health(|| Health::new(90)),  Movement(|| Movement::new(3.)),  AggroRadius(|| AggroRadius(5.)),  UnitType(|| UnitType::Air),  UnitState(|| UnitState::Moving),  Attack(|| Attack::new(AttackType::Melee(80), AttackTargetType::All, 1., 2.)),  Hitbox(|| Hitbox(0.5)),  StunnedTimer, )] struct Bat;  fn spawn\_bat(  trigger: Trigger<SpawnBat>,  mut server: ResMut<QuinnetServer>,  mut cmd: Commands, ) {  let &SpawnBat(pos, owner) = trigger.event();   let entity = cmd.spawn((Bat, pos, owner)).id();   server  .endpoint\_mut()  .broadcast\_message\_on(  ServerChannel::OrderedReliable,  ServerMessage::SpawnUnit {  server\_entity: entity,  unit: Unit::Bat,  pos,  owner,  },  )  .unwrap(); } |

|  |
| --- |
| server\src\units\bomber.rs |
| use bevy::prelude::\*; use bevy\_quinnet::server::QuinnetServer; use common::{  ArenaPos, Health, PlayerNumber, Projectile, ServerChannel, ServerMessage, Unit, UnitState, };  use crate::ai::{AggroRadius, Attack, AttackTargetType, AttackType, Movement, StunnedTimer};  use super::{Hitbox, UnitType};  pub(super) fn plugin(app: &mut App) {  app.add\_observer(spawn\_bomber); }  #[derive(Event)] pub struct SpawnBomber(pub ArenaPos, pub PlayerNumber);  #[derive(Component)] #[require(  Health(|| Health::new(230)),  Movement(|| Movement::new(2.)),  AggroRadius(|| AggroRadius(5.5)),  UnitType(|| UnitType::Ground),  UnitState(|| UnitState::Moving),  Attack(|| Attack::new(AttackType::Ranged(Projectile::Bomb),  AttackTargetType::Ground, 0.7, 4.5)),  Hitbox(|| Hitbox(0.5)),  StunnedTimer, )] struct Bomber;  fn spawn\_bomber(  trigger: Trigger<SpawnBomber>,  mut server: ResMut<QuinnetServer>,  mut cmd: Commands, ) {  let &SpawnBomber(pos, owner) = trigger.event();   let entity = cmd.spawn((Bomber, pos, owner)).id();   server  .endpoint\_mut()  .broadcast\_message\_on(  ServerChannel::OrderedReliable,  ServerMessage::SpawnUnit {  server\_entity: entity,  unit: Unit::Bomber,  pos,  owner,  },  )  .unwrap(); } |

|  |
| --- |
| server\src\units\giant.rs |
| use bevy::prelude::\*; use bevy\_quinnet::server::QuinnetServer; use common::{ArenaPos, Health, PlayerNumber, ServerChannel, ServerMessage, Unit, UnitState};  use crate::ai::{Movement, StunnedTimer};  use super::{Hitbox, UnitType};  pub(super) fn plugin(app: &mut App) {  app.add\_observer(spawn\_giant);   app.add\_systems(FixedUpdate, update\_giants); }  #[derive(Event)] pub struct SpawnGiant(pub ArenaPos, pub PlayerNumber);  #[derive(Component)] #[require(  Health(|| Health::new(800)),  Movement(|| Movement::new(1.5)),  UnitType(|| UnitType::Ground),  UnitState(|| UnitState::Moving),  Hitbox(|| Hitbox(1.)),  StunnedTimer, )] pub struct Giant {  pub target: Option<Entity>,  pub attack\_range: f32,  pub damage: u16,  pub cooldown: Timer, }  fn spawn\_giant(  trigger: Trigger<SpawnGiant>,  mut server: ResMut<QuinnetServer>,  mut cmd: Commands, ) {  let &SpawnGiant(pos, owner) = trigger.event();   let entity = cmd  .spawn((  Giant {  target: None,  attack\_range: 2.,  damage: 120,  cooldown: Timer::from\_seconds(1.5, TimerMode::Repeating),  },  pos,  owner,  ))  .id();   server  .endpoint\_mut()  .broadcast\_message\_on(  ServerChannel::OrderedReliable,  ServerMessage::SpawnUnit {  server\_entity: entity,  unit: Unit::Giant,  pos,  owner,  },  )  .unwrap(); }  fn update\_giants(  mut giants: Query<(  &mut Giant,  &mut UnitState,  &mut Movement,  &ArenaPos,  &PlayerNumber,  )>,  mut towers: Query<(Entity, &ArenaPos, &mut Health, &PlayerNumber), Without<Movement>>,  time: Res<Time>, ) {  for (mut giant, mut state, mut movement, pos, player\_num) in &mut giants {  match \*state {  UnitState::Idle => panic!("Гигант не может находиться в UnitState::Idle"),  UnitState::Moving => {  if let Some(target) = movement.target {  let Ok((tower, tower\_pos, \_, \_)) = towers.get(target) else {  continue;  };   if pos.distance(tower\_pos) <= giant.attack\_range {  \*state = UnitState::Attacking;  giant.target = Some(tower);  }  continue;  };   let mut closest\_tower = None;  let mut minimal\_distance = 1000.;  for (tower, tower\_pos, \_, tower\_player\_num) in &towers {  if player\_num == tower\_player\_num {  continue;  }  let distance = pos.distance(tower\_pos);  if distance < minimal\_distance {  closest\_tower = Some(tower);  minimal\_distance = distance;  }  }  movement.target = closest\_tower;  }  UnitState::Attacking => {  if let Some(target) = giant.target {  if !giant.cooldown.tick(time.delta()).just\_finished() {  continue;  }   let Ok((\_, \_, mut health, \_)) = towers.get\_mut(target) else {  giant.target = None;  \*state = UnitState::Moving;  continue;  };  health.0 = health.0.saturating\_sub(giant.damage);  continue;  }  }  }  } } |

|  |
| --- |
| server\src\units\king\_tower.rs |
| use bevy::prelude::\*; use bevy\_quinnet::server::QuinnetServer; use common::{  ArenaPos, Health, PlayerNumber, Projectile, ServerChannel, ServerMessage, Unit, UnitState, };  use crate::ai::{Attack, AttackTargetType, AttackType};  use super::{Hitbox, UnitType};  pub(super) fn plugin(app: &mut App) {  app.add\_observer(spawn\_king\_tower); }  #[derive(Event)] pub struct SpawnKingTower(pub ArenaPos, pub PlayerNumber);  #[derive(Component)] #[require(  Health(|| Health::new(2400)),  UnitType(|| UnitType::Ground),  UnitState,  Attack(|| Attack::new(AttackType::Ranged(Projectile::Fireball),  AttackTargetType::All, 1., 6.)),  Hitbox(|| Hitbox(2.)), )] struct KingTower;  fn spawn\_king\_tower(  trigger: Trigger<SpawnKingTower>,  mut server: ResMut<QuinnetServer>,  mut cmd: Commands, ) {  let &SpawnKingTower(pos, owner) = trigger.event();   let entity = cmd.spawn((KingTower, pos, owner)).id();   server  .endpoint\_mut()  .broadcast\_message\_on(  ServerChannel::OrderedReliable,  ServerMessage::SpawnUnit {  server\_entity: entity,  unit: Unit::KingTower,  pos,  owner,  },  )  .unwrap(); } |

|  |
| --- |
| server\src\units\mod.rs |
| use archer\_tower::SpawnArcherTower; use bat::SpawnBat; use bevy::prelude::\*; use bomber::SpawnBomber; use common::{ArenaPos, PlayerNumber, Unit}; use giant::SpawnGiant; use king\_tower::SpawnKingTower; use musketeer::SpawnMusketeer; use priest::SpawnPriest; use rus::SpawnRus; pub use giant::Giant;  mod archer\_tower; mod bat; mod bomber; mod giant; mod king\_tower; mod musketeer; mod priest; mod rus;  pub(super) fn plugin(app: &mut App) {  app.add\_plugins((  archer\_tower::plugin,  king\_tower::plugin,  rus::plugin,  musketeer::plugin,  bat::plugin,  priest::plugin,  bomber::plugin,  giant::plugin,  )); }  #[derive(Component)] pub enum UnitType {  Air,  Ground, }  #[derive(Component)] pub struct Hitbox(pub f32);  pub(super) trait SpawnUnit {  fn spawn(&self, pos: ArenaPos, player\_num: PlayerNumber, cmd: &mut Commands); }  impl SpawnUnit for Unit {  fn spawn(&self, pos: ArenaPos, player\_num: PlayerNumber, cmd: &mut Commands) {  match self {  Unit::ArcherTower => cmd.trigger(SpawnArcherTower(pos, player\_num)),  Unit::KingTower => cmd.trigger(SpawnKingTower(pos, player\_num)),  Unit::Rus => cmd.trigger(SpawnRus(pos, player\_num)),  Unit::Musketeer => cmd.trigger(SpawnMusketeer(pos, player\_num)),  Unit::Bat => cmd.trigger(SpawnBat(pos, player\_num)),  Unit::Priest => cmd.trigger(SpawnPriest(pos, player\_num)),  Unit::Bomber => cmd.trigger(SpawnBomber(pos, player\_num)),  Unit::Giant => cmd.trigger(SpawnGiant(pos, player\_num)),  }  } } |

|  |
| --- |
| server\src\units\musketeer.rs |
| use bevy::prelude::\*; use bevy\_quinnet::server::QuinnetServer; use common::{  ArenaPos, Health, PlayerNumber, Projectile, ServerChannel, ServerMessage, Unit, UnitState, };  use crate::ai::{AggroRadius, Attack, AttackTargetType, AttackType, Movement, StunnedTimer};  use super::{Hitbox, UnitType};  pub(super) fn plugin(app: &mut App) {  app.add\_observer(spawn\_musketeer); }  #[derive(Event)] pub struct SpawnMusketeer(pub ArenaPos, pub PlayerNumber);  #[derive(Component)] #[require(  Health(|| Health::new(340)),  Movement(|| Movement::new(2.)),  AggroRadius(|| AggroRadius(7.)),  UnitType(|| UnitType::Ground),  UnitState(|| UnitState::Moving),  Attack(|| Attack::new(AttackType::Ranged(Projectile::Bullet),  AttackTargetType::All, 0.75, 6.)),  Hitbox(|| Hitbox(0.5)),  StunnedTimer, )] struct Musketeer;  fn spawn\_musketeer(  trigger: Trigger<SpawnMusketeer>,  mut server: ResMut<QuinnetServer>,  mut cmd: Commands, ) {  let &SpawnMusketeer(pos, owner) = trigger.event();   let entity = cmd.spawn((Musketeer, pos, owner)).id();   server  .endpoint\_mut()  .broadcast\_message\_on(  ServerChannel::OrderedReliable,  ServerMessage::SpawnUnit {  server\_entity: entity,  unit: Unit::Musketeer,  pos,  owner,  },  )  .unwrap(); } |

|  |
| --- |
| server\src\units\priest.rs |
| use bevy::prelude::\*; use bevy\_quinnet::server::QuinnetServer; use common::{  ArenaPos, Health, PlayerNumber, Projectile, ServerChannel, ServerMessage, Unit, UnitState, };  use crate::ai::{AggroRadius, Attack, AttackTargetType, AttackType, Movement, StunnedTimer};  use super::{Hitbox, UnitType};  pub(super) fn plugin(app: &mut App) {  app.add\_observer(spawn\_priest); }  #[derive(Event)] pub struct SpawnPriest(pub ArenaPos, pub PlayerNumber);  #[derive(Component)] #[require(  Health(|| Health::new(400)),  Movement(|| Movement::new(2.)),  AggroRadius(|| AggroRadius(7.)),  UnitType(|| UnitType::Ground),  UnitState(|| UnitState::Moving),  Attack(|| Attack::new(AttackType::Ranged(Projectile::Fireball),  AttackTargetType::All, 0.75, 6.)),  Hitbox(|| Hitbox(0.5)),  StunnedTimer, )] struct Priest;  fn spawn\_priest(  trigger: Trigger<SpawnPriest>,  mut server: ResMut<QuinnetServer>,  mut cmd: Commands, ) {  let &SpawnPriest(pos, owner) = trigger.event();   let entity = cmd.spawn((Priest, pos, owner)).id();   server  .endpoint\_mut()  .broadcast\_message\_on(  ServerChannel::OrderedReliable,  ServerMessage::SpawnUnit {  server\_entity: entity,  unit: Unit::Priest,  pos,  owner,  },  )  .unwrap(); } |

|  |
| --- |
| server\src\units\rus.rs |
| use bevy::prelude::\*; use bevy\_quinnet::server::QuinnetServer; use common::{ArenaPos, Health, PlayerNumber, ServerChannel, ServerMessage, Unit, UnitState};  use crate::ai::{AggroRadius, Attack, AttackTargetType, AttackType, Movement, StunnedTimer};  use super::{Hitbox, UnitType};  pub(super) fn plugin(app: &mut App) {  app.add\_observer(spawn\_rus); }  #[derive(Event)] pub struct SpawnRus(pub ArenaPos, pub PlayerNumber);  #[derive(Component)] #[require(  Health(|| Health::new(690)),  Movement(|| Movement::new(2.)),  AggroRadius(|| AggroRadius(5.)),  UnitType(|| UnitType::Ground),  UnitState(|| UnitState::Moving),  Attack(|| Attack::new(AttackType::Melee(80), AttackTargetType::Ground, 0.8, 2.)),  Hitbox(|| Hitbox(0.5)),  StunnedTimer, )] struct Rus;  fn spawn\_rus(  trigger: Trigger<SpawnRus>,  mut server: ResMut<QuinnetServer>,  mut cmd: Commands, ) {  let &SpawnRus(pos, owner) = trigger.event();   let entity = cmd.spawn((Rus, pos, owner)).id();   server  .endpoint\_mut()  .broadcast\_message\_on(  ServerChannel::OrderedReliable,  ServerMessage::SpawnUnit {  server\_entity: entity,  unit: Unit::Rus,  pos,  owner,  },  )  .unwrap(); } |