GAME DEVELOPMENT WITH RUST

Simon Potzernheim

12/2024

WHY RUST?

RUST MAKES CERTAIN (BAD) PATTERNS MORE PAINFUL THAN OTHERS, WHICH IS A GOOD THING!

- The easiest for Rust are very often the easiest in general
- I had to learn good patterns the hard way, without Rust's help
- For games, one of these is ECS design, but there are OTHERS
- Rust rewards data-oriented design with clear ownership

Catherine West

PERSONAL REASONS TO CHOSE RUST FOR GAME DEVELOPMENT

- Code first
- No Heavy Editor
- Easy setup and build and learn incrementally
- My favorite language & ecosystem
- Great community on discord
- Flexible Target Platforms (Browser, Windows, Linux, etc)

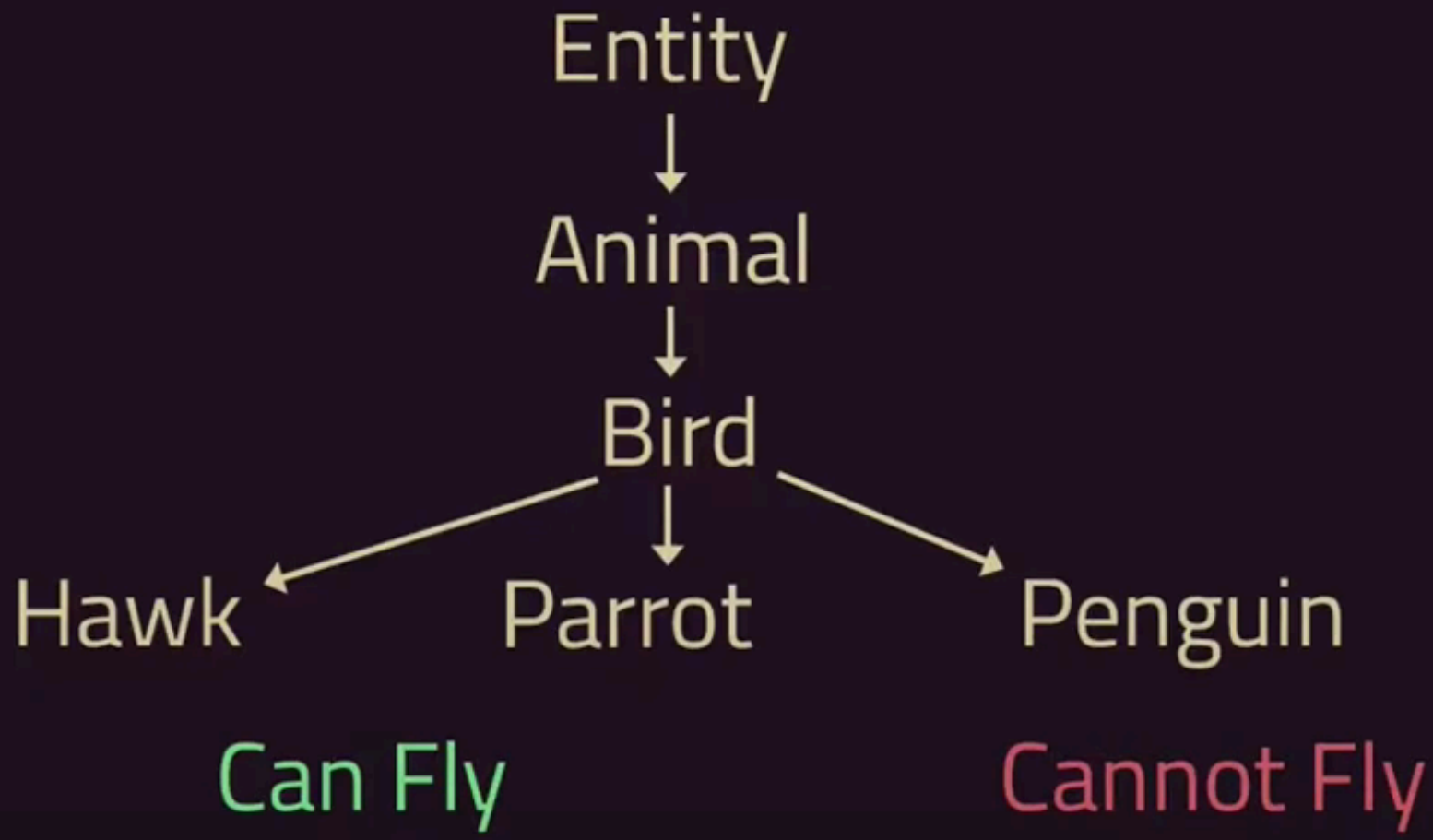
Simon

WHAT IS ECS?

How many accessors could you possibly need?

```
class Player :
  public virtual ToolUserEntity,
  public virtual LoungingEntity,
  public virtual ChattyEntity,
  public virtual DamageBarEntity,
  public virtual PortraitEntity,
  public virtual NametagEntity,
  public virtual PhysicsEntity,
  public virtual EmoteEntity {

public:
  Player(PlayerConfigPtr config, Uuid uuid = Uuid());
  Player(PlayerConfigPtr config, Json const& diskStore);
  Player(PlayerConfigPtr config, ByteArray const& netStore);

  ClientContextPtr clientContext() const;
  void setClientContext(ClientContextPtr clientContext);

  StatisticsPtr statistics() const;
  void setStatistics(StatisticsPtr statistics);

  QuestManagerPtr questManager() const;

  Json diskStore();
  ByteArray netStore();

  EntityType entityType() const override;

  void init(World* world, EntityId entityId, EntityMode mode) override;
  void uninit() override;

  Vec2F position() const override;
  Vec2F velocity() const override;

  Vec2F mouthPosition() const override;
  Vec2F mouthOffset() const;
  Vec2F feetOffset() const;
```

OBJECT HIERARCHY



THE ECS WAY 1/2

- Entity - things in the world
- Component - Data of Entities
- System - update enitity data

THE ECS WAY 2/3

```
let entity = world.spawn()
.insert(Animal)
.insert(CanFly)
.insert(CanWalk)
.insert(AnimalSpecies::Hawk)
.insert(AnimalType::Bird)
.id();
```

Bevy 0.7 - 07/2022

THE ECS WAY 3/3

```
let entity = world.spawn(
  (
    Animal,
    CanFly,
    CanWalk,
    AnimalSpecies::Hawk,
    AnimalType::Bird
  )
  .id();
```

Bevy 0.15 - 12/2024

GOAL: AVOID OOP PRACTISES

- Avoid Inheritance
- Avoid Composition
- Avoid Polymorphism
- Avoid Complexity

CORE STRENGTHS

- Flexibility
- Reasonability
- Composability
- Performance (Rendering)

ENTITY

- Entity is just an id existing in the world

```
let mut world = World::new(); // This is a engine Resource
let entity: Entity = world.spawn((components go here)).id();
```

```
let mut map: WorldMap = WorldMap::new(); // This is a game Res
map.add(entity);
```

ENTITY CAN HAVE CHILDREN

This is a logical hierarchy. The ECS itself is a flat data structure.

```
menu.with_children(|parent| {
  parent.spawn(
    (
      Text::new("Play"),
      TextFont {
          font: asset_server.load("fonts/pixeled.ttf"),
          font_size: 40.0,
      },
      TextColor(Color::rgb(0.9, 0.9, 0.9)),
      Button,
      BackGroundColor(Color::rgb(0.1, 0.1, 0.1)),
    )
  );
})
```

BEVY WORLD

```rust
pub struct World {
    id: WorldId,
    pub(crate) entities: Entities,
    pub(crate) components: Components,
    pub(crate) archetypes: Archetypes,
    pub(crate) storages: Storages,
    pub(crate) bundles: Bundles,
    /// new in 2024:
    pub(crate) observers: Observers,
    ...

}
```

COMPONENT EXAMPLES

- Position / Coordinates
- Sprite
- Charakter Health in an RPG

COMPONENT EXAMPLES

Ofen Components are just markers

```rust
#[derive(Component, Default)]
struct SecondWindowCamera3d;
```

```rust
#[derive(Component)]
pub struct Coordinates {
    pub x: u16,
    pub y: u16,
}
```

(because of systems)

SYSTEM EXAMPLE

```rust
fn hello_world() {
  println!("hello world!");
}
pub fn hide_popup(mut commands: Commands, popup: Res<PopupRef>
  // despawns a subtree of entities
  commands.entity(popup.0).despawn_recursive();
  // removes a resource there is only one per type per world
  commands.remove_resource::<PopupRef>()
}
```

SYSTEM QUERIES

```
fn menu(
  mut state: ResMut<State<AppState>>,// <- this is a state res
  mut interaction_query: Query<
      (&Interaction, &mut UiColor), // <- this are components
      (Changed<Interaction>, With<Button>),//<- more component
      >,
) {
    interaction_query.iter() // <- this is a iterator
    ...
}
```

SYSTEM QUERIES

```
for (interaction, mut color) in interaction_query.iter_mut() {
  match *interaction {
      Interaction::Clicked => {
          *color = PRESSED_BUTTON.into();
          state.set(AppState::InGame).unwrap();
      }
      Interaction::Hovered => {
          *color = HOVERED_BUTTON.into();
      }
      Interaction::None => {
          *color = NORMAL_BUTTON.into();
      }
  }
}
```

BEVY GITHUB

LIVE EXAMPLES

https://bevyengine.org/examples

LIVE EXAMPLES

Clone the Bevy repo:

```
git clone https://github.com/bevyengine/bevy
cd bevy
git checkout latest
git checkout v0.7.0 <- 2022
git checkout v0.15.0 <- 2024
```

Try the examples in the examples folder

```
cargo run --example breakout
```

MY TETRIS DEMO

https://simon-an.github.io/bevy-tetris/

recently updated to Bevy 0.15 (will get some updates soon)

BEVY ENGINE CORE

DEV TOOLS AND EDITORS FOR BEVY

- bevy_inspector_egui editor-like inspector window
- bevy_editor_pls is an editor-like interface for fly camera, performance diagnostics, inspector panels.
- bevy_mod_debugdump is a tool to help visualize your App Schedule (systems and stages)
- bevy_lint checks your Bevy code for some common issues.
- bevycheck translates compiler errors to user-friendly Bevy-specific error messages

COORDINATE SYSTEM

- Bevy uses a right-handed Y-up coordinate system.
- Bevy uses the same coordinate system for 3D, 2D, and UI, for consistency.

TRANSFORMS

- a Transform is what allows you to place an object in the game world. It is a combination of the object's "translation" (position/coordinates), "rotation", and "scale" (size adjustment).
- You move objects around by modifying the translation, rotate them by modifying the rotation, and make them larger or smaller by modifying the scale.

HIERARCHICAL (PARENT/CHILD) ENTITIES

- Technically, the Entities/Components cannot form a hierarchy (the ECS is a flat data structure).
- Logical hierarchies are a common pattern in games.
- Bevy supports creating logical link between entities, by simply adding Parent and Children components on the respective entities.
- Commands has methods for adding children to entities, which automatically add the correct components.

BEVY ENGINE USAGE

APP / MAIN.RS

```rust
use bevy::prelude::*;

fn main() {
    App::new().run();
}
```

```
cargo run
```

PLUGINS 1/2

- used to organize bevy features
- used to organize a game
- used to create libraries for bevy

```
fn main() {
App::new()
    // This already creates a window
    // because it contains Core, Input and Window Plugin
    .add_plugins(DefaultPlugins)
    .run();
}
```

PLUGINS 2/2

```
struct MyPlugin;

impl Plugin for MyPlugin {
    fn build(&self, app: &mut App) {
        app
            .init_resource::<MyOtherResource>()
            .add_event::<MyEvent>()
            .add_startup_system(plugin_init)
            .add_system(my_system);
    }
}

fn main() {
    App::new()
        .add_plugins(DefaultPlugins)
```

RESOURCES

Entities and Components are great for representing complex, query-able groups of data. But most Apps will also require "globally unique" data of some kind. In Bevy ECS, we represent globally unique data using Resources.

- Elapsed Time
- Asset Collections (sounds, textures, meshes)
- Renderers

RESOURCES

```
impl Plugin for HelloPlugin {
    fn build(&self, app: &mut App) {
        // the reason we call from_seconds with the true
        // flag is to make the timer repeat itself
        app.insert_resource(
          GreetTimer(Timer::from_seconds(2.0, true))
        )
    }
}
```

- Resources can be added globally or in systems

SYSTEMS

```
fn greet_people(
  time: Res<Time>, query: Query<&Name, With<Person>>
) {
    for name in query.iter() {
        println!("hello {}!", name.0);
    }
}
```

- Systems are called by the Engine each frame
- Parameters of System come from the ECS automatically

SYSTEMS: LOCAL RESOURCES

```rust
#[derive(Default)]
struct MyState;

fn my_system1(mut local: Local<MyState>) {
    // you can do anything you want with the local here
}
```

- Local is managed by ECS automatically
- ECS creates an instance by calling Default::default()

EXPLICIT SYSTEM ORDERING <- DEPRECATED

```
App::new()
  // order doesn't matter for these systems:
  .add_system(particle_effects)
  .add_system(npc_behaviors)
  .add_system(enemy_movement)
  .add_system(input_handling)
  .add_system(
    player_movement
    // `player_movement` must always run before `enemy_movemen
    .before(enemy_movement)
    // `player_movement` must always run after `input_handling
    .after(input_handling)
  )
  .run();
```

- Systems can be ordered with .before and .after

SYSTEM SETS <- DEPRECATED

```
App::new()
.add_plugins(DefaultPlugins)
// group our input handling systems into a set
.add_system_set(
    SystemSet::new()
        .label("input")
        .with_system(keyboard_input)
        .with_system(gamepad_input)
)
// our "net" systems should run before "input"
.add_system_set(
    SystemSet::new()
        .label("net")
        .before("input")
        // individual systems can still have
```

SYSTEM CHAINING

```
fn main() {
  App::new()
        // ...
        .add_system(net_receive.chain(handle_io_errors))
        // ...
        .run();
}
```

- handle_io_errors is called with the result of net_receive

LIFECYCLE/STAGES 1/2

All systems to be run by Bevy are contained in stages. Every frame update, Bevy executes each stage, in order. Within each stage, Bevy's scheduling algorithm can run many systems in parallel, using multiple CPU cores for good performance.

LIFECYCLE/STAGES 2/2

```
pub enum StartupStage {
  PreStartup,
  Startup,
  PostStartup,
}
pub enum CoreStage {
  First,
  PreUpdate,
  Update,
  PostUpdate,
  Last,
}
```

CUSTOM STAGE

```rust
fn main() {
  // label for our debug stage
  static DEBUG: &str = "debug";

  App::new()
      .add_plugins(DefaultPlugins)

      // add DEBUG stage after Bevy's Update
      // also make it single-threaded
      .add_stage_after(CoreStage::Update, DEBUG, SystemStage::

      // systems are added to the `CoreStage::Update` stage by
      .add_system(player_gather_xp)
      .add_system(player_take_damage)
```

STATES - ONLY FOR PRO'S :D

```rust
#[derive(Debug, Clone, Eq, PartialEq, Hash)]
enum AppState {
    MainMenu,
    InGame,
    Paused,
}

fn main() {
    App::new()
        .add_plugins(DefaultPlugins)

        // add the app state type
        .add_state(AppState::MainMenu)

        // add systems to run regardless of state, as usual
```

QUERIES AND CHANGE DETECTION

```rust
/// Print the stats of friendly players when they change
fn debug_stats_change(
    query: Query<
        // components
        (&Health, &PlayerXp),
        // filters
        (Without<Enemy>, Or<(Changed<Health>, Changed<PlayerXp
    >,
) {
    for (health, xp) in query.iter() {
        eprintln!(
            "hp: {}+{}, xp: {}",
            health.hp, health.extra, xp.0
        );
    }
```

- Change detection is done with a simple query in the systems for

INPUT

```rust
fn mouse_button_input(
    buttons: Res<Input<MouseButton>>,
) {
    if buttons.just_pressed(MouseButton::Left) {
        // Left button was pressed
    }
    if buttons.just_released(MouseButton::Left) {
        // Left Button was released
    }
    if buttons.pressed(MouseButton::Right) {
        // Right Button is being held down
    }
    // we can check multiple at once with `.any_*`
    if buttons.any_just_pressed([MouseButton::Left, MouseButto
        // Either the left or the right button was just presse
```

COMMANDS 1/2

- Use Commands to spawn/despawn entities, add/remove components on existing entities, manage resources.
- These actions do not take effect immediately; they are queued to be performed later when it is safe to do so.
- That means your other systems will see them on the next frame update

COMMANDS 2/2

```rust
fn spawn_player(
    mut commands: Commands,
) {
  // manage resources
  commands.insert_resource(GoalsReached { main_goal: false, bo
  commands.remove_resource::<MyResource>();
  // create a new entity using `spawn`
  let entity_id = commands.spawn()
      // add a component
      .insert(ComponentA)
      // add a bundle
      .insert_bundle(MyBundle::default())
      // get the Entity ID
      .id();
  // shorthand for creating an entity with a bundle
```

EVENTS

```rust
struct LevelUpEvent(Entity);

fn player_level_up(
    mut ev_levelup: EventWriter<LevelUpEvent>,
    query: Query<(Entity, &PlayerXp)>,
) {
    for (entity, xp) in query.iter() {
        if xp.0 > 1000 {
            ev_levelup.send(LevelUpEvent(entity));
        }
    }
}

fn debug_levelups(
    mut ev_levelup: EventReader<LevelUpEvent>,
```

ENTITIES AND RENDERING

```
pub(crate) fn spawn_tetromino(
  mut commands: Commands,
  board: Res<Board>,
  mut spawn_event_rdr: EventReader<SpawnEvent>,
  mut game_over_event_wtr: EventWriter<GameOverEvent>,
) {
  let translation = Vec3::new(......);
  for event in spawn_event_rdr.iter() {
    // ...
    commands.entity(board.entity).with_children(|mut parent| {
      for block in blocks.into_iter() {
        // ...
        let entity = parent
        .spawn()
        .insert_bundle(bevy_sprite::bundle::SpriteBundle {
```

TIME

- The Time resource is your main global source of timing information
- Can be accessed from any system
- You should derive all timings from it
- Bevy updates these values at the beginning of every frame

TIMERS 1/3

```rust
fn asteroids_fly(
    time: Res<Time>,
    mut q: Query<&mut Transform, With<Asteroid>>,
) {
    for mut transform in q.iter_mut() {
        // move our asteroids along the X axis
        // at a speed of 10.0 units per second
        transform.translation.x += 10.0 * time.delta_seconds()
    }
}
```

TIMERS 2/3

```rust
use std::time::Instant;

/// Say, for whatever reason, we want to keep track
/// of when exactly some specific entities were spawned.
#[derive(Component)]
struct SpawnedTime(Instant);

fn spawn_my_stuff(
    mut commands: Commands,
    time: Res<Time>,
) {
    commands.spawn()
        .insert(SpawnedTime(time.startup() + time.time_since_s
}
```

TIMERS 3/3

```
#[derive(Component)]
struct FuseTime {
    /// track when the bomb should explode (non-repeating time
    timer: Timer,
}

mut commands: Commands,
mut q: Query<(Entity, &mut FuseTime)>,
time: Res<Time>,
) {
    for (entity, mut fuse_timer) in q.iter_mut() {
        // timers gotta be ticked, to work
        fuse_timer.timer.tick(time.delta());

        // if it finished, despawn the bomb
```

TESTS

```
fn headless_mode() {
  App::new()
       .insert_resource(WgpuSettings {
           backends: None,
           ..Default::default()
       })
       .add_plugins_with(DefaultPlugins, |group| group.disable:
       .add_plugin(EguiPlugin)
       .update();
}
```

EGUI

- Popular Multiplatfrom / Target UI Crate
- Used in many crates and plugins in the bevy ecosystem
- Can easily be used to create a game ui

CONCLUSION

- Don't confuse OOP with "Data Oriented Design"
- Getting into Game Development with Rust is easy
- Unity and Unreal Engine removed from personal TODO list
- ECS + Rust feals like a natural fit
- Game Development + Rust feals like a natural fit
- More to come (3D in Space)

RESOURCES

- ECS: https://www.youtube.com/channel/UCZzs3Umh6sRiYS_IQIQD
- Using RUST FOR GAME DEVELOPMENT: https://youtu.be/oHYs-UqS458
- https://github.com/bevyengine
- https://bevyengine.org/learn/book/introduction/
- https://bevy-cheatbook.github.io/

CODING SCORE IMPLEMENTATION

TODO