My experience of using Rust for game development

Vlad Zhukov

Content

Game

- 3D world with 2D graphics and 2D gamplay
- Slices of 3D world with 2D mechanics

Project structure

Main crate structure

```
crate
 engine
     events
     game_logic
     graphics
        ...
     map
        ...
     npc
        ...
     physics
     ui
         ...
```

Graphics

Glium with specs

```
// gfx_system.rs
#[repr(transparent)]
pub struct Renderer<'a> {
    pub data: RenderingData<'a>,
           // &RenderingData is often used as a parameter
}

impl<'a, 'b> System<'a> for Renderer<'b> {
    ...
  fn run(&mut self, data: Self::SystemData) {
    let mut target = self.data.display.draw();
                          // display: &'a glium::Display,
    <DRAW CALLS HERE>
    <CONROD GUI EVENTS HANDLING(AND DRAWING)>
    ...
    target.finish().unwrap();
  }
  ..
}
```

```rust
pub struct RenderingData<'a> {
  pub positions_buffers: Vec<glium::VertexBuffer<Vertex3>>,
  pub normals_buffers: Vec<glium::VertexBuffer<Normal3>>,
  pub indices_buffers: Vec<glium::IndexBuffer<u16>>,
  pub textures: Vec<glium::texture::SrgbTexture2d>,
  ...
}
}
```

it's sort of "my little ECS" while using specs...

If any of these please let me know

- mb there is way to implement thread local components/resources with specs
- mb there is better way to do it

```
// main.rs
let mut rendering_system = Renderer::new(..)
let mut dispatcher = DispatcherBuilder::new()
  .with(..)
  ...
  .with_thread_local(rendering_system)
```

glium can be called only thread locally

Physics

- nphysics
- Many systems
- One of these calls step()

How do I use nphysics with specs?

Solution by thiolliere airjump-multi(GitHub)

```rust
//retained_storage.rs
pub struct RetainedStorage<C, T = UnprotectedStorage<C>> {
    retained: Vec<C>,
    storage: T,
    phantom: PhantomData<C>,
}

impl<C, T> Retained<C> for RetainedStorage<C, T> {
    fn retained(&mut self) -> Vec<C> {
        mem::replace(&mut self.retained, vec![])
    }
}
//shared_physics.rs
pub struct RigidBody(BodyHandle);
impl ::specs::Component for RigidBody {
    type Storage = RetainedStorage<Self>;
}
```

Solution by thiolliere airjump-multi(GitHub)

```rust
// shared_physics.rs
pub fn safe_insert<'a>(
    entity: ::specs::Entity,
    // position, inertia, ...
    bodies_handle: &mut ::specs::WriteStorage<'a, ::component::RigidBody>,
    physic_world: &mut ::resource::PhysicWorld,
    bodies_map: &mut ::resource::BodiesMap,
) -> Self {
    let body_handle = physic_world.add_rigid_body(position, inertia ...);
    bodies_map.insert(body_handle, entity);
    bodies_handle.insert(entity, RigidBody(body_handle));
    RigidBody(body_handle)
}
```

Solution by thiolliere airjump-multi(GitHub)

```rust
    // main.rs
    pub fn safe_maintain(world: &mut specs::World) {
      world.maintain();
      let mut physic_world = world.write_resource::<World<f32>>();
      let mut bodies_map = world.write_resource::<BodiesMap>();

      let retained = world
          .write_storage::<PhysicComponent>()
          .retained()
          .iter()
          .map(|r| r.body_handle)
          .collect::<Vec<_>>();
      physic_world.remove_bodies(&retained);
      for handle in &retained {
          bodies_map.remove(handle);
      }
  }
```

UI

- conrod
- small ”ingame” UI(dragging objects for now)

conrod is easy, let’s talk about ingame UI

ImGUI style (only for editor, easiest implementation)

```rust
// shared_game_logic.rs
#[derive(Default, Debug)]
pub struct IngameUI {
    pub current_draw_primitives: Vec<DrawPrimitive>,
    pub mouse_position: (f32, f32),
    // * shift between mouse end object position when starting moving
    pub moving_shift: Option<Vector3<f32>>,
    // * position of mouse in space
    pub moving_position: Option<Vector3<f32>>,
                      // Option easier for impl default
    // * state of UI entities interation
    pub hover_id: Option<usize>,
    pub selected_id: Option<usize>,
    pub moving_id: Option<usize>,

    // * state of entities interction
    pub entity_hover_id: Option<::specs::Entity>,
    pub entity_selected_id: Option<::specs::Entity>,
    pub entity_moving_id: Option<::specs::Entity>,
}
```

```rust
#[derive(Debug)]
pub enum DrawPrimitive {
    Cube{
        position: Vector3<f32>,
        size: f32,
    },
}

pub struct Draggable {
    pub position: Vector3<f32>,
}
```

```
// ingame_ui.rs
let draggable = Draggable::new(
    Vector3::new(
        current_target_position.x,
        current_target_position.y,
        current_target_position.z)
);
match draggable.set(
    draggable_id as usize,
    ingame_ui, // &mut IngameUI
    & system_resources,
    &camera,
    mouse_position
) {
    Some(new_position) => {
        // DO THOMETHING WITH NEW VALUE
    }
    None => ()
}
```

Map and saving

How to implement saving?

- specs serde searializing deserializing (Hard)
- searializing/deserializing intermidiate objects (Ok)

map.ron $\leftrightarrow$ Map with RawEntityes $\leftrightarrow$ specs objects

```
// map/common.rs
#[derive(Debug, Serialize, Deserialize)]
pub struct Map {
    pub entities: Vec<RawEntity>,
    pub positions: Vec<Option<Isometry3<f32>>>,
    pub scales: Vec<Option<ScaleComponent>>,
    pub slices: Slices,
}
```

```rust
// map/common.rs
#[derive(Debug, Default, Clone, Serialize, Deserialize, Builder)]
#[builder(setter(into))]
#[builder(default)]
#[builder(derive(Debug, Serialize, Deserialize))]
pub struct RawEntity {
    pub gl_name: Option<String>,
    pub cs_texture_name: Option<String>,
    pub physic_name: Option<String>,
    pub renderer_name: Option<String>,
    ...
}
```

Builder pattern with *derive_builder* crate

Compilation time

Rust nightly opt-level. My crate is in debug, dependencies are in release

```
[profile.dev]
opt-level = 0
debug = true

[profile.dev.overrides."*"]
opt-level = 3
```

Separate project on crates helps(not that much)
My crates are:

- main crate
- misc (probably will name it "common")
- physics
- rendering

$\sim$ 10s when change parent crate vs $\sim$ 20s without crate separating
Don't do it like that