```
// ********************************************************************************
// ********** tutorial1 - main
use sdl2::event::Event;
use sdl2::keyboard::Keycode;

use std::time::Duration;

fn main() -> Result<(), String> {
    println!("Starting Rusteroids");

    let sdl_context = sdl2::init()?;
    let video_subsystem = sdl_context.video()?;

    let _window = video_subsystem
        .window("Rusteroids", 800, 600)
        .position_centered()
        .build()
        .expect("could not initialize video subsystem");

    let mut event_pump = sdl_context.event_pump()?;

    'running: loop {
        // Handle events
        for event in event_pump.poll_iter() {
            match event {
                Event::Quit { .. } => {
                    break 'running;
                }
                Event::KeyDown {
                    keycode: Some(Keycode::Escape),
                    ..
                } => {
                    break 'running;
                }
                _ => {}
            }
        }

        // Time management
        ::std::thread::sleep(Duration::new(0, 1_000_000_000u32 / 60));
    }

    Ok(())
}
```

```
// ********************************************************************************
// ********** tutorial2 - main
use sdl2::event::Event;
use sdl2::keyboard::Keycode;
use sdl2::pixels::Color;
use sdl2::rect::Rect;
use sdl2::render::{TextureCreator, WindowCanvas};
use sdl2::video::WindowContext;

use std::path::Path;
use std::time::Duration;

fn render(
    canvas: &mut WindowCanvas,
    texture_creator: &TextureCreator<WindowContext>,
    font: &sdl2::ttf::Font,
) -> Result<(), String> {
    //
    let color = Color::RGB(0, 0, 0);
    canvas.set_draw_color(color);
    canvas.clear();

    // Draw Greeting
    let hello_text: String = "Hello World".to_string();
    let surface = font
        .render(&hello_text)
        .blended(Color::RGBA(255, 0, 0, 128))
        .map_err(|e| e.to_string())?;

    let texture = texture_creator
        .create_texture_from_surface(&surface)
        .map_err(|e| e.to_string())?;

    let target = Rect::new(10 as i32, 0 as i32, 200 as u32, 100 as u32);
    canvas.copy(&texture, None, Some(target))?;

    canvas.present();
    Ok(())
}

fn main() -> Result<(), String> {
    println!("Starting Rusteroids");

    let sdl_context = sdl2::init()?;
    let video_subsystem = sdl_context.video()?;

    let window = video_subsystem
        .window("Rusteroids", 800, 600)
        .position_centered()
        .build()
        .expect("could not initialize video subsystem");

    let mut canvas = window
        .into_canvas()
        .build()
        .expect("could not make a canvas");

    let texture_creator = canvas.texture_creator();

    // Prepare fonts
    let ttf_context = sdl2::ttf::init().map_err(|e| e.to_string())?;
    let font_path: &Path = Path::new(&"fonts/OpenSans-Bold.ttf");
    let mut font = ttf_context.load_font(font_path, 128)?;
    font.set_style(sdl2::ttf::FontStyle::BOLD);

    let mut event_pump = sdl_context.event_pump()?;

    'running: loop {
        // Handle events
        for event in event_pump.poll_iter() {
            match event {
                Event::Quit { .. } => {
                    break 'running;
                }
                Event::KeyDown {
                    keycode: Some(Keycode::Escape),
                    ..
                } => {
                    break 'running;
                }
                _ => {}
            }
        }

        render(&mut canvas, &texture_creator, &font)?;

        // Time management
        ::std::thread::sleep(Duration::new(0, 1_000_000_000u32 / 60));
    }

    Ok(())
}
```

```
// *********************************************************************************
// ********** tutorial3 - main
use sdl2::event::Event;
use sdl2::keyboard::Keycode;
use sdl2::pixels::Color;
use sdl2::render::{TextureCreator, WindowCanvas};
use sdl2::video::WindowContext;

use sdl2::rect::Point;
use sdl2::rect::Rect;

use std::path::Path;
use std::time::Duration;

pub mod texture_manager;

const IMAGE_WIDTH: u32 = 100;
const IMAGE_HEIGHT: u32 = 100;
const OUTPUT_WIDTH: u32 = 50;
const OUTPUT_HEIGHT: u32 = 50;
const SCREEN_WIDTH: i32 = 800;
const SCREEN_HEIGHT: i32 = 600;

fn render(
    canvas: &mut WindowCanvas,
    texture_manager: &mut texture_manager::TextureManager<WindowContext>,
    _texture_creator: &TextureCreator<WindowContext>,
    _font: &sdl2::ttf::Font,
) -> Result<(), String> {
    //
    let color = Color::RGB(0, 0, 0);
    canvas.set_draw_color(color);
    canvas.clear();

    // Draw Space Ship
    let src = Rect::new(0, 0, IMAGE_WIDTH, IMAGE_HEIGHT);
    let x: i32 = (SCREEN_WIDTH / 2) as i32;
    let y: i32 = (SCREEN_HEIGHT / 2) as i32;

    let dest = Rect::new(
        x - ((OUTPUT_WIDTH / 2) as i32),
        y - ((OUTPUT_HEIGHT / 2) as i32),
        OUTPUT_WIDTH,
        OUTPUT_HEIGHT,
    );
    let center = Point::new((OUTPUT_WIDTH / 2) as i32, (OUTPUT_HEIGHT) as i32);

    let texture = texture_manager.load("img/space_ship.png")?;

    canvas.copy_ex(
        &texture, // Texture object
        src,      // source rect
        dest,     // destination rect
        279.0,    // angle (degrees)
        center,   // center
        false,    // flip horizontal
        false,    // flip vertical
    )?;

    canvas.present();
    Ok(())
}

fn main() -> Result<(), String> {
    println!("Starting Rusteroids");

    let sdl_context = sdl2::init()?;
    let video_subsystem = sdl_context.video()?;

    let window = video_subsystem
        .window("Rusteroids", 800, 600)
        .position_centered()
        .build()
        .expect("could not initialize video subsystem");

    let mut canvas = window
        .into_canvas()
        .build()
        .expect("could not make a canvas");

    let texture_creator = canvas.texture_creator();
    let mut tex_man = texture_manager::TextureManager::new(&texture_creator);

    // Load the images before the main loop so we don't try and load during gameplay
    tex_man.load("img/space_ship.png")?;

    // Prepare fonts
    let ttf_context = sdl2::ttf::init().map_err(|e| e.to_string())?;
    let font_path: &Path = Path::new(&"fonts/OpenSans-Bold.ttf");
    let mut font = ttf_context.load_font(font_path, 128)?;
    font.set_style(sdl2::ttf::FontStyle::BOLD);

    let mut event_pump = sdl_context.event_pump()?;

    'running: loop {
        // Handle events
        for event in event_pump.poll_iter() {
            match event {
                Event::Quit { .. } => {
                    break 'running;
                }
                Event::KeyDown {
                    keycode: Some(Keycode::Escape),
                    ..
                } => {
                    break 'running;
                }
                _ => {}
            }
        }

         render(&mut canvas, &mut tex_man, &texture_creator, &font)?;

        // Time management
        ::std::thread::sleep(Duration::new(0, 1_000_000_000u32 / 60));
    }

    Ok(())
}

// ********** tutorial3 - texture_manager
use sdl2::image::LoadTexture;
use sdl2::render::{Texture, TextureCreator};
use std::borrow::Borrow;
use std::collections::HashMap;
use std::hash::Hash;
use std::rc::Rc;

pub type TextureManager<'l, T> = ResourceManager<'l, String, Texture<'l>, TextureCreator<T>>;

pub struct ResourceManager<'l, K, R, L>
where
    K: Hash + Eq,
    L: 'l + ResourceLoader<'l, R>,
{
    loader: &'l L,
    cache: HashMap<K, Rc<R>>,
}

impl<'l, K, R, L> ResourceManager<'l, K, R, L>
where
    K: Hash + Eq,
    L: ResourceLoader<'l, R>,
{
    pub fn new(loader: &'l L) -> Self {
        ResourceManager {
            cache: HashMap::new(),
            loader: loader,
        }
    }

    // Generics magic to allow a HashMap to use String as a key
    // while allowing it to use &str for gets
    pub fn load<D>(&mut self, details: &D) -> Result<Rc<R>, String>
    where
        L: ResourceLoader<'l, R, Args = D>,
        D: Eq + Hash + ?Sized,
        K: Borrow<D> + for<'a> From<&'a D>,
    {
        self.cache.get(details).cloned().map_or_else(
            || {
                let resource = Rc::new(self.loader.load(details)?);
                self.cache.insert(details.into(), resource.clone());
                Ok(resource)
            },
            Ok,
        )
    }
}

// Generic trait to Load any Resource Kind
pub trait ResourceLoader<'l, R> {
    type Args: ?Sized;
    fn load(&'l self, data: &Self::Args) -> Result<R, String>;
}

// TextureCreator knows how to load Textures
impl<'l, T> ResourceLoader<'l, Texture<'l>> for TextureCreator<T> {
    type Args = str;
    fn load(&'l self, path: &str) -> Result<Texture, String> {
        // println!("LOADED A TEXTURE");
        self.load_texture(path)
    }
}
```

```
// ******************************************************************************
// ********** tutorial4 - main
use sdl2::event::Event;
use sdl2::keyboard::Keycode;
use sdl2::pixels::Color;
use sdl2::render::{TextureCreator, WindowCanvas};
use sdl2::video::WindowContext;

use sdl2::rect::Point;
use sdl2::rect::Rect;

use std::collections::HashMap;
use std::path::Path;
use std::time::Duration;

pub mod texture_manager;
pub mod utils;

const IMAGE_WIDTH: u32 = 100;
const IMAGE_HEIGHT: u32 = 100;
const OUTPUT_WIDTH: u32 = 100;
const OUTPUT_HEIGHT: u32 = 100;
const SCREEN_WIDTH: i32 = 800;
const SCREEN_HEIGHT: i32 = 600;

fn render(
    canvas: &mut WindowCanvas,
    texture_manager: &mut texture_manager::TextureManager<WindowContext>,
    _texture_creator: &TextureCreator<WindowContext>,
    _font: &sdl2::ttf::Font,
    key_manager: &HashMap<String, bool>,
) -> Result<(), String> {
    //
    let color = Color::RGB(0, 0, 0);
    canvas.set_draw_color(color);
    canvas.clear();

    // Draw Space Ship
    let src = Rect::new(0, 0, IMAGE_WIDTH, IMAGE_HEIGHT);
    let x: i32 = (SCREEN_WIDTH / 2) as i32;
    let y: i32 = (SCREEN_HEIGHT / 2) as i32;

    let dest = Rect::new(
        x - ((OUTPUT_WIDTH / 2) as i32),
        y - ((OUTPUT_HEIGHT / 2) as i32),
        OUTPUT_WIDTH,
        OUTPUT_HEIGHT,
    );
    let center = Point::new((OUTPUT_WIDTH / 2) as i32, (OUTPUT_HEIGHT / 2) as i32);

    let texture = texture_manager.load("img/space_ship.png")?;
    let mut angle: f64 = 0.0;

    if utils::is_key_pressed(&key_manager, "W") {
        angle = 0.0;
    } else if utils::is_key_pressed(&key_manager, "D") {
        angle = 90.0;
    } else if utils::is_key_pressed(&key_manager, "S") {
        angle = 180.0;
    } else if utils::is_key_pressed(&key_manager, "A") {
        angle = 270.0;
    }

    canvas.copy_ex(
        &texture, // Texture object
        src,      // source rect
        dest,     // destination rect
        angle,    // angle (degrees)
        center,   // center
        false,    // flip horizontal
        false,    // flip vertical
    )?;

    canvas.present();
    Ok(())
}

fn main() -> Result<(), String> {
    println!("Starting Rusteroids");

    let sdl_context = sdl2::init()?;
    let video_subsystem = sdl_context.video()?;

    let window = video_subsystem
        .window("Rusteroids", 800, 600)
        .position_centered()
        .build()
        .expect("could not initialize video subsystem");

    let mut canvas = window
        .into_canvas()
        .build()
        .expect("could not make a canvas");

    let texture_creator = canvas.texture_creator();
    let mut tex_man = texture_manager::TextureManager::new(&texture_creator);

    // Load the images before the main loop so we don't try and load during gameplay
    tex_man.load("img/space_ship.png")?;

    // Prepare fonts
    let ttf_context = sdl2::ttf::init().map_err(|e| e.to_string())?;
    let font_path: &Path = Path::new(&"fonts/OpenSans-Bold.ttf");
    let mut font = ttf_context.load_font(font_path, 128)?;
    font.set_style(sdl2::ttf::FontStyle::BOLD);

    let mut event_pump = sdl_context.event_pump()?;
    let mut key_manager: HashMap<String, bool> = HashMap::new();

    'running: loop {
        // Handle events
        for event in event_pump.poll_iter() {
            match event {
                Event::Quit { .. } => {
                    break 'running;
                }
                Event::KeyDown {
                    keycode: Some(Keycode::Escape),
                    ..
                } => {
                    break 'running;
                }
                Event::KeyDown { keycode, .. } => match keycode {
                    None => {}
                    Some(key) => {
                        utils::key_down(&mut key_manager, key.to_string());
                    }
                },
                Event::KeyUp { keycode, .. } => match keycode {
                    None => {}
                    Some(key) => {
                        utils::key_up(&mut key_manager, key.to_string());
                    }
                },
                _ => {}
            }
        }

        render(
            &mut canvas,
            &mut tex_man,
            &texture_creator,
            &font,
            &key_manager,
        )?;

        // Time management
        ::std::thread::sleep(Duration::new(0, 1_000_000_000u32 / 60));
    }

    Ok(())
}
```

// ********** tutorial4 - texture_manager

// ********** tutorial4 - utils

```
use std::collections::HashMap;

// Key Manager Functions
pub fn key_down(key_manager: &mut HashMap<String, bool>, keyname: String) {
    if !key_manager.contains_key(&keyname) {
        key_manager.entry(keyname).or_insert(true);
    } else {
        if let Some(x) = key_manager.get_mut(&keyname) {
            *x = true;
        }
    }
}

pub fn key_up(key_manager: &mut HashMap<String, bool>, keyname: String) {
    if !key_manager.contains_key(&keyname) {
        key_manager.entry(keyname).or_insert(false);
    } else {
        if let Some(x) = key_manager.get_mut(&keyname) {
            *x = false;
        }
    }
}

pub fn is_key_pressed(key_manager: &HashMap<String, bool>, value: &str) -> bool {
    key_manager.contains_key(&value.to_string())
        && key_manager.get(&value.to_string()) == Some(&true)
}
```

```
// ********************************************************************************
// ********** tutorial5 - main
use sdl2::event::Event;
use sdl2::keyboard::Keycode;
use sdl2::pixels::Color;
use sdl2::render::{TextureCreator, WindowCanvas};
use sdl2::video::WindowContext;

use sdl2::rect::Point;
use sdl2::rect::Rect;

use specs::{Join, World, WorldExt};

use std::collections::HashMap;
use std::path::Path;
use std::time::Duration;

pub mod components;
pub mod texture_manager;
pub mod utils;

pub mod game;

fn render(
    canvas: &mut WindowCanvas,
    texture_manager: &mut texture_manager::TextureManager<WindowContext>,
    _texture_creator: &TextureCreator<WindowContext>,
    _font: &sdl2::ttf::Font,
    ecs: &World,
) -> Result<(), String> {
    //
    let color = Color::RGB(0, 0, 0);
    canvas.set_draw_color(color);
    canvas.clear();

    let positions = ecs.read_storage::<components::Position>();
    let renderables = ecs.read_storage::<components::Renderable>();

    for (renderable, pos) in (&renderables, &positions).join() {
        let src = Rect::new(0, 0, renderable.i_w, renderable.i_h);
        let x: i32 = pos.x as i32;
        let y: i32 = pos.y as i32;
        let dest = Rect::new(
            x - ((renderable.o_w / 2) as i32),
            y - ((renderable.o_h / 2) as i32),
            renderable.o_w,
            renderable.o_h,
        );

        let center = Point::new((renderable.o_w / 2) as i32, (renderable.o_h / 2) as i32);
        let texture = texture_manager.load(&renderable.tex_name)?;
        canvas.copy_ex(
            &texture, src,     //source rect
            dest,    // dest rect
            pos.rot, // angle
            center,  // center
            false,   // flip horizontal
            false,   // flip vertical
        )?;
    }

    canvas.present();
    Ok(())
}

struct State {
    ecs: World,
}

fn main() -> Result<(), String> {
    println!("Starting Rusteroids");

    let sdl_context = sdl2::init()?;
    let video_subsystem = sdl_context.video()?;

    let window = video_subsystem
        .window("Rusteroids", 800, 600)
        .position_centered()
        .build()
        .expect("could not initialize video subsystem");

    let mut canvas = window
        .into_canvas()
        .build()
        .expect("could not make a canvas");

    let texture_creator = canvas.texture_creator();
    let mut tex_man = texture_manager::TextureManager::new(&texture_creator);

    // Load the images before the main loop so we don't try and load during gameplay
    tex_man.load("img/space_ship.png")?;

    // Prepare fonts
    let ttf_context = sdl2::ttf::init().map_err(|e| e.to_string())?;
    let font_path: &Path = Path::new(&"fonts/OpenSans-Bold.ttf");
    let mut font = ttf_context.load_font(font_path, 128)?;
    font.set_style(sdl2::ttf::FontStyle::BOLD);

    let mut event_pump = sdl_context.event_pump()?;
    let mut key_manager: HashMap<String, bool> = HashMap::new();

    let mut gs = State { ecs: World::new() };
    gs.ecs.register::<components::Position>();
    gs.ecs.register::<components::Renderable>();
    gs.ecs.register::<components::Player>();

    game::load_world(&mut gs.ecs);

    'running: loop {
        // Handle events
        for event in event_pump.poll_iter() {
            match event {
                Event::Quit { .. } => {
                    break 'running;
                }
                Event::KeyDown {
                    keycode: Some(Keycode::Escape),
                    ..
                } => {
                    break 'running;
                }
                Event::KeyDown { keycode, .. } => match keycode {
                    None => {}
                    Some(key) => {
                        utils::key_down(&mut key_manager, key.to_string());
                    }
                },
                Event::KeyUp { keycode, .. } => match keycode {
                    None => {}
                    Some(key) => {
                        utils::key_up(&mut key_manager, key.to_string());
                    }
                },
                _ => {}
            }
        }
        game::update(&mut gs.ecs, &mut key_manager);
        render(&mut canvas, &mut tex_man, &texture_creator, &font, &gs.ecs)?;

        // Time management
        ::std::thread::sleep(Duration::new(0, 1_000_000_000u32 / 60));
    }

    Ok(())
}
```

// ********** tutorial5 - components

```
use specs::prelude::*;
use specs_derive::Component;

#[derive(Component)]
pub struct Position {
    pub x: f64,
    pub y: f64,
    pub rot: f64,
}

#[derive(Component)]
pub struct Renderable {
    pub tex_name: String,
    pub i_w: u32,
    pub i_h: u32,
    pub o_w: u32,
    pub o_h: u32,
    pub frame: u32,
    pub total_frames: u32,
    pub rot: f64,
}

#[derive(Component)]
pub struct Player {}
```

// ********** tutorial5 - game

```
use specs::Join;
use specs::{Builder, World, WorldExt};
use std::collections::HashMap;

const ROTATION_SPEED: f64 = 1.5;

pub fn update(ecs: &mut World, key_manager: &mut HashMap<String, bool>) {
    let mut positions = ecs.write_storage::<crate::components::Position>();
    let players = ecs.read_storage::<crate::components::Player>();

    for (_, pos) in (&players, &mut positions).join() {
        if crate::utils::is_key_pressed(&key_manager, "D") {
            pos.rot += ROTATION_SPEED;
        }
        if crate::utils::is_key_pressed(&key_manager, "A") {
            pos.rot -= ROTATION_SPEED;
        }
        if pos.rot > 360.0 {
            pos.rot -= 360.0;
        }
        if pos.rot < 0.0 {
            pos.rot += 360.0;
        }
    }
}

pub fn load_world(ecs: &mut World) {
    ecs.create_entity()
        .with(crate::components::Position {
            x: 350.0,
            y: 250.0,
            rot: 0.0,
        })
        .with(crate::components::Renderable {
            tex_name: String::from("img/space_ship.png"),
            i_w: 100,
            i_h: 100,
            o_w: 100,
            o_h: 100,
            frame: 0,
            total_frames: 1,
            rot: 0.0,
        })
        .with(crate::components::Player {})
        .build();
}
```

```
// ********** tutorial5 - texture_manager
```

```
// ********** tutorial5 - utils
```

```rust
// ***********************************************************************************
// ********** tutorial6 - main
use sdl2::event::Event;
use sdl2::keyboard::Keycode;
use sdl2::pixels::Color;
use sdl2::render::{TextureCreator, WindowCanvas};
use sdl2::video::WindowContext;

use sdl2::rect::Point;
use sdl2::rect::Rect;

use specs::{Join, World, WorldExt};

use std::collections::HashMap;
use std::path::Path;
use std::time::Duration;

pub mod components;
pub mod texture_manager;
pub mod utils;

pub mod game;

const GAME_WIDTH: u32 = 800;
const GAME_HEIGHT: u32 = 600;

fn render(
    canvas: &mut WindowCanvas,
    texture_manager: &mut texture_manager::TextureManager<WindowContext>,
    _texture_creator: &TextureCreator<WindowContext>,
    _font: &sdl2::ttf::Font,
    ecs: &World,
) -> Result<(), String> {
    //
    let color = Color::RGB(0, 0, 0);
    canvas.set_draw_color(color);
    canvas.clear();

    let positions = ecs.read_storage::<components::Position>();
    let renderables = ecs.read_storage::<components::Renderable>();

    for (renderable, pos) in (&renderables, &positions).join() {
        let src = Rect::new(0, 0, renderable.i_w, renderable.i_h);
        let x: i32 = pos.x as i32;
        let y: i32 = pos.y as i32;
        let dest = Rect::new(
            x - ((renderable.o_w / 2) as i32),
            y - ((renderable.o_h / 2) as i32),
            renderable.o_w,
            renderable.o_h,
        );

        let center = Point::new((renderable.o_w / 2) as i32, (renderable.o_h / 2) as i32);
        let texture = texture_manager.load(&renderable.tex_name)?;
        canvas.copy_ex(
            &texture, src,     //source rect
            dest,    // dest rect
            pos.rot, // angle
            center,  // center
            false,   // flip horizontal
            false,   // flip vertical
        )?;
    }

    canvas.present();
    Ok(())
}

struct State {
    ecs: World,
}

fn main() -> Result<(), String> {
    println!("Starting Rusteroids");

    let sdl_context = sdl2::init()?;
    let video_subsystem = sdl_context.video()?;

    let window = video_subsystem
         .window("Rusteroids", GAME_WIDTH, GAME_HEIGHT)
        .position_centered()
        .build()
        .expect("could not initialize video subsystem");

    let mut canvas = window
        .into_canvas()
        .build()
        .expect("could not make a canvas");

    let texture_creator = canvas.texture_creator();
    let mut tex_man = texture_manager::TextureManager::new(&texture_creator);

    // Load the images before the main loop so we don't try and load during gameplay
    tex_man.load("img/space_ship.png")?;

    // Prepare fonts
    let ttf_context = sdl2::ttf::init().map_err(|e| e.to_string())?;
    let font_path: &Path = Path::new(&"fonts/OpenSans-Bold.ttf");
    let mut font = ttf_context.load_font(font_path, 128)?;
    font.set_style(sdl2::ttf::FontStyle::BOLD);

    let mut event_pump = sdl_context.event_pump()?;
    let mut key_manager: HashMap<String, bool> = HashMap::new();

    let mut gs = State { ecs: World::new() };
    gs.ecs.register::<components::Position>();
    gs.ecs.register::<components::Renderable>();
    gs.ecs.register::<components::Player>();

    game::load_world(&mut gs.ecs);

    'running: loop {
        // Handle events
        for event in event_pump.poll_iter() {
            match event {
                Event::Quit { .. } => {
                    break 'running;
                }
                Event::KeyDown {
                    keycode: Some(Keycode::Escape),
                    ..
                } => {
                    break 'running;
                }
                Event::KeyDown { keycode, .. } => match keycode {
                    None => {}
                    Some(key) => {
                        utils::key_down(&mut key_manager, key.to_string());
                    }
                },
                Event::KeyUp { keycode, .. } => match keycode {
                    None => {}
                    Some(key) => {
                        utils::key_up(&mut key_manager, key.to_string());
                    }
                },
                _ => {}
            }
        }
        game::update(&mut gs.ecs, &mut key_manager);
        render(&mut canvas, &mut tex_man, &texture_creator, &font, &gs.ecs)?;

        // Time management
        ::std::thread::sleep(Duration::new(0, 1_000_000_000u32 / 60));
    }

    Ok(())
}
```

// ********** tutorial6 - components

// ********** tutorial6 - game

```
use specs::Join;
use specs::{Builder, World, WorldExt};
use std::collections::HashMap;

const ROTATION_SPEED: f64 = 1.5;
const PLAYER_SPEED: f64 = 4.5;

pub fn update(ecs: &mut World, key_manager: &mut HashMap<String, bool>) {
    let mut positions = ecs.write_storage::<crate::components::Position>();
    let players = ecs.read_storage::<crate::components::Player>();

    for (_, pos) in (&players, &mut positions).join() {
        if crate::utils::is_key_pressed(&key_manager, "D") {
            pos.rot += ROTATION_SPEED;
        }
        if crate::utils::is_key_pressed(&key_manager, "A") {
            pos.rot -= ROTATION_SPEED;
        }

        if crate::utils::is_key_pressed(&key_manager, "W") {
            let radians = pos.rot.to_radians();

            pos.x += PLAYER_SPEED * radians.sin();
            pos.y -= PLAYER_SPEED * radians.cos();
        }

        if pos.rot > 360.0 {
            pos.rot -= 360.0;
        }
        if pos.rot < 0.0 {
            pos.rot += 360.0;
        }

        if pos.x > crate::GAME_WIDTH.into() {
            pos.x -= crate::GAME_WIDTH as f64;
        }
        if pos.x < 0.0 {
            pos.x += crate::GAME_WIDTH as f64;
        }
        if pos.y > crate::GAME_HEIGHT.into() {
            pos.y -= crate::GAME_HEIGHT as f64;
        }
        if pos.y < 0.0 {
            pos.y += crate::GAME_HEIGHT as f64;
        }
    }
}

pub fn load_world(ecs: &mut World) {
    ecs.create_entity()
        .with(crate::components::Position {
            x: 350.0,
            y: 250.0,
            rot: 0.0,
        })
        .with(crate::components::Renderable {
            tex_name: String::from("img/space_ship.png"),
            i_w: 100,
            i_h: 100,
            o_w: 100,
            o_h: 100,
            frame: 0,
            total_frames: 1,
            rot: 0.0,
        })
        .with(crate::components::Player {})
        .build();
}
```

// ********** tutorial6 - texture_manager

// ********** tutorial6 - utils

```
// ***************************************************************************
// ********** tutorial7 - main

// ********** tutorial7 - components
use specs::prelude::*;
use specs_derive::Component;
use vector2d::Vector2D;

#[derive(Component)]
pub struct Position {
    pub x: f64,
    pub y: f64,
    pub rot: f64,
}

#[derive(Component)]
pub struct Renderable {
    pub tex_name: String,
    pub i_w: u32,
    pub i_h: u32,
    pub o_w: u32,
    pub o_h: u32,
    pub frame: u32,
    pub total_frames: u32,
    pub rot: f64,
}

#[derive(Component)]
pub struct Player {
    pub impulse: Vector2D<f64>,   // The next impulse to add to the speed
    pub cur_speed: Vector2D<f64>, // The current speed of the player
}

// ********** tutorial7 - game
use specs::{Builder, Join, World, WorldExt};
use std::collections::HashMap;
use vector2d::Vector2D;

const ROTATION_SPEED: f64 = 1.5;
const PLAYER_SPEED: f64 = 4.5;

pub fn update(ecs: &mut World, key_manager: &mut HashMap<String, bool>) {
    let mut positions = ecs.write_storage::<crate::components::Position>();
    let mut players = ecs.write_storage::<crate::components::Player>();

    for (player, pos) in (&mut players, &mut positions).join() {
        if crate::utils::is_key_pressed(&key_manager, "D") {
            pos.rot += ROTATION_SPEED;
        }
        if crate::utils::is_key_pressed(&key_manager, "A") {
            pos.rot -= ROTATION_SPEED;
        }
        update_movement(pos, player);
        if crate::utils::is_key_pressed(&key_manager, "W") {
            let radians = pos.rot.to_radians();

            let move_x = PLAYER_SPEED * radians.sin();
            let move_y = PLAYER_SPEED * radians.cos();
            let move_vec = Vector2D::<f64>::new(move_x, move_y);

            player.impulse += move_vec;
        }

        if pos.rot > 360.0 {
            pos.rot -= 360.0;
        }
        if pos.rot < 0.0 {
            pos.rot += 360.0;
        }

        if pos.x > crate::GAME_WIDTH.into() {
            pos.x -= crate::GAME_WIDTH as f64;
        }
        if pos.x < 0.0 {
            pos.x += crate::GAME_WIDTH as f64;
        }
        if pos.y > crate::GAME_HEIGHT.into() {
            pos.y -= crate::GAME_HEIGHT as f64;
        }
        if pos.y < 0.0 {
            pos.y += crate::GAME_HEIGHT as f64;
        }
    }
}

const MAX_SPEED: f64 = 3.5;
const FRICTION: f64 = 0.99;

pub fn update_movement(
    pos: &mut crate::components::Position,
    player: &mut crate::components::Player,
) {
    player.cur_speed *= FRICTION;

    player.cur_speed += player.impulse;
    if player.cur_speed.length() > MAX_SPEED {
        player.cur_speed = player.cur_speed.normalise();
        player.cur_speed = player.cur_speed * MAX_SPEED;
    }

    pos.x += player.cur_speed.x;
    pos.y -= player.cur_speed.y;

    player.impulse = vector2d::Vector2D::new(0.0, 0.0);
}

pub fn load_world(ecs: &mut World) {
    ecs.create_entity()
```

```
        .with(crate::components::Position {
            x: 350.0,
            y: 250.0,
            rot: 0.0,
        })
        .with(crate::components::Renderable {
            tex_name: String::from("img/space_ship.png"),
            i_w: 100,
            i_h: 100,
            o_w: 100,
            o_h: 100,
            frame: 0,
            total_frames: 1,
            rot: 0.0,
        })
         .with(crate::components::Player {
             impulse: vector2d::Vector2D::new(0.0, 0.0),
             cur_speed: vector2d::Vector2D::new(0.0, 0.0),
         })
        .build();
}
```

```
// ********** tutorial7 - texture_manager
```

```
// ********** tutorial7 - utils
// ********************************************************************************
// ********** tutorial8 - main
use sdl2::event::Event;
use sdl2::keyboard::Keycode;
use sdl2::pixels::Color;
use sdl2::render::{TextureCreator, WindowCanvas};
use sdl2::video::WindowContext;

use sdl2::rect::Point;
use sdl2::rect::Rect;

use specs::{DispatcherBuilder, Join, World, WorldExt};

use std::collections::HashMap;
use std::path::Path;
use std::time::Duration;

pub mod asteroid;
pub mod components;
pub mod game;
pub mod texture_manager;
pub mod utils;

const GAME_WIDTH: u32 = 800;
const GAME_HEIGHT: u32 = 600;

fn render(
    canvas: &mut WindowCanvas,
    texture_manager: &mut texture_manager::TextureManager<WindowContext>,
    _texture_creator: &TextureCreator<WindowContext>,
    _font: &sdl2::ttf::Font,
    ecs: &World,
) -> Result<(), String> {
    //
    let color = Color::RGB(0, 0, 0);
    canvas.set_draw_color(color);
    canvas.clear();

    let positions = ecs.read_storage::<components::Position>();
    let renderables = ecs.read_storage::<components::Renderable>();

    for (renderable, pos) in (&renderables, &positions).join() {
        let src = Rect::new(0, 0, renderable.i_w, renderable.i_h);
        let x: i32 = pos.x as i32;
        let y: i32 = pos.y as i32;
        let dest = Rect::new(
            x - ((renderable.o_w / 2) as i32),
            y - ((renderable.o_h / 2) as i32),
            renderable.o_w,
            renderable.o_h,
        );

        let center = Point::new((renderable.o_w / 2) as i32, (renderable.o_h / 2) as i32);
        let texture = texture_manager.load(&renderable.tex_name)?;
        canvas.copy_ex(
            &texture,
            src,           //source rect
            dest,          // dest rect
              renderable.rot, // angle
            center,        // center
            false,         // flip horizontal
            false,         // flip vertical
        )?;
    }

    canvas.present();
    Ok(())
}

struct State {
    ecs: World,
}

fn main() -> Result<(), String> {
    println!("Starting Rusteroids");

    let sdl_context = sdl2::init()?;
    let video_subsystem = sdl_context.video()?;

    let window = video_subsystem
        .window("Rusteroids", GAME_WIDTH, GAME_HEIGHT)
        .position_centered()
        .build()
        .expect("could not initialize video subsystem");

    let mut canvas = window
        .into_canvas()
        .build()
        .expect("could not make a canvas");

    let texture_creator = canvas.texture_creator();
    let mut tex_man = texture_manager::TextureManager::new(&texture_creator);

    // Load the images before the main loop so we don't try and load during gameplay
    tex_man.load("img/space_ship.png")?;
     tex_man.load("img/asteroid.png")?;

    // Prepare fonts
    let ttf_context = sdl2::ttf::init().map_err(|e| e.to_string())?;
    let font_path: &Path = Path::new(&"fonts/OpenSans-Bold.ttf");
    let mut font = ttf_context.load_font(font_path, 128)?;
    font.set_style(sdl2::ttf::FontStyle::BOLD);

    let mut event_pump = sdl_context.event_pump()?;
    let mut key_manager: HashMap<String, bool> = HashMap::new();

    let mut gs = State { ecs: World::new() };
    gs.ecs.register::<components::Position>();
    gs.ecs.register::<components::Renderable>();
    gs.ecs.register::<components::Player>();
     gs.ecs.register::<components::Asteroid>();

     let mut dispatcher = DispatcherBuilder::new()
         .with(asteroid::AsteroidMover, "asteroid_mover", &[])
        .build();

    game::load_world(&mut gs.ecs);

    'running: loop {
        // Handle events
        for event in event_pump.poll_iter() {
            match event {
                Event::Quit { .. } => {
                    break 'running;
                }
                Event::KeyDown {
                    keycode: Some(Keycode::Escape),
                    ..
                } => {
                    break 'running;
                }
                Event::KeyDown { keycode, .. } => match keycode {
                    None => {}
                    Some(key) => {
                        utils::key_down(&mut key_manager, key.to_string());
                    }
                },
                Event::KeyUp { keycode, .. } => match keycode {
                    None => {}
                    Some(key) => {
                        utils::key_up(&mut key_manager, key.to_string());
                    }
                },
                _ => {}
            }
        }

        game::update(&mut gs.ecs, &mut key_manager);
         dispatcher.dispatch(&gs.ecs);
        render(&mut canvas, &mut tex_man, &texture_creator, &font, &gs.ecs)?;

        // Time management
        ::std::thread::sleep(Duration::new(0, 1_000_000_000u32 / 60));
    }

    Ok(())
}
```

// ********** tutorial8 - asteroid

```
use specs::{Join, System, WriteStorage};

pub struct AsteroidMover;

use crate::components;

impl<'a> System<'a> for AsteroidMover {
    type SystemData = (
        WriteStorage<'a, components::Position>,
        WriteStorage<'a, components::Renderable>,
        WriteStorage<'a, components::Asteroid>,
    );

    fn run(&mut self, mut data: Self::SystemData) {
        for (pos, rend, asteroid) in (&mut data.0, &mut data.1, &data.2).join() {
            let radians = pos.rot.to_radians();

            pos.x += asteroid.speed * radians.sin();
            pos.y -= asteroid.speed * radians.cos();

            let half_width = (rend.o_w / 2) as u32;
            let half_height = (rend.o_h / 2) as u32;

            if pos.x > (crate::GAME_WIDTH - half_width).into() || pos.x < half_width.into() {
                pos.rot = 360.0 - pos.rot;
            } else if pos.y > (crate::GAME_HEIGHT - half_height).into()
                || pos.y < half_height.into()
            {
                if pos.rot > 180.0 {
                    pos.rot = 540.0 - pos.rot;
                } else {
                    pos.rot = 180.0 - pos.rot;
                }
            }

            rend.rot += asteroid.rot_speed;
            if rend.rot > 360.0 {
                rend.rot -= 360.0;
            }
            if rend.rot < 0.0 {
                rend.rot += 360.0;
            }
        }
    }
}
```

// ********** tutorial8 - components

```
use specs::prelude::*;
use specs_derive::Component;
use vector2d::Vector2D;

#[derive(Component)]
pub struct Position {
    pub x: f64,
    pub y: f64,
    pub rot: f64,
}


#[derive(Component)]
pub struct Renderable {
    pub tex_name: String,
    pub i_w: u32,
    pub i_h: u32,
    pub o_w: u32,
    pub o_h: u32,
    pub frame: u32,
    pub total_frames: u32,
    pub rot: f64,
}

#[derive(Component)]
pub struct Player {
    pub impulse: Vector2D<f64>,   // The next impulse to add to the speed
    pub cur_speed: Vector2D<f64>, // The current speed of the player
}

#[derive(Component)]
pub struct Asteroid {
    pub speed: f64,
    pub rot_speed: f64,
}
```

```
// ********** tutorial8 - game
use specs::{Builder, Join, World, WorldExt};
use std::collections::HashMap;
use vector2d::Vector2D;

const ROTATION_SPEED: f64 = 1.5;
const PLAYER_SPEED: f64 = 4.5;

pub fn update(ecs: &mut World, key_manager: &mut HashMap<String, bool>) {
    let mut positions = ecs.write_storage::<crate::components::Position>();
    let mut players = ecs.write_storage::<crate::components::Player>();
    let mut renderables = ecs.write_storage::<crate::components::Renderable>();

    for (player, pos, renderable) in (&mut players, &mut positions, &mut renderables).join() {
        if crate::utils::is_key_pressed(&key_manager, "D") {
            pos.rot += ROTATION_SPEED;
        }
        if crate::utils::is_key_pressed(&key_manager, "A") {
            pos.rot -= ROTATION_SPEED;
        }
        update_movement(pos, player);
        if crate::utils::is_key_pressed(&key_manager, "W") {
            let radians = pos.rot.to_radians();

            let move_x = PLAYER_SPEED * radians.sin();
            let move_y = PLAYER_SPEED * radians.cos();
            let move_vec = Vector2D::<f64>::new(move_x, move_y);

            player.impulse += move_vec;
        }

        if pos.rot > 360.0 {
            pos.rot -= 360.0;
        }
        if pos.rot < 0.0 {
            pos.rot += 360.0;
        }

        if pos.x > crate::GAME_WIDTH.into() {
            pos.x -= crate::GAME_WIDTH as f64;
        }
        if pos.x < 0.0 {
            pos.x += crate::GAME_WIDTH as f64;
        }
        if pos.y > crate::GAME_HEIGHT.into() {
            pos.y -= crate::GAME_HEIGHT as f64;
        }
        if pos.y < 0.0 {
            pos.y += crate::GAME_HEIGHT as f64;
        }

        // Update the graphic to reflect the rotation
        renderable.rot = pos.rot;
    }
}

const MAX_SPEED: f64 = 3.5;
const FRICTION: f64 = 0.99;

pub fn update_movement(
    pos: &mut crate::components::Position,
    player: &mut crate::components::Player,
) {
    player.cur_speed *= FRICTION;

    player.cur_speed += player.impulse;
    if player.cur_speed.length() > MAX_SPEED {
        player.cur_speed = player.cur_speed.normalise();
        player.cur_speed = player.cur_speed * MAX_SPEED;
    }

    pos.x += player.cur_speed.x;
    pos.y -= player.cur_speed.y;

    player.impulse = vector2d::Vector2D::new(0.0, 0.0);
}

pub fn load_world(ecs: &mut World) {
    ecs.create_entity()
        .with(crate::components::Position {
            x: 350.0,
            y: 250.0,
            rot: 0.0,
        })
        .with(crate::components::Renderable {
            tex_name: String::from("img/space_ship.png"),
            i_w: 100,
            i_h: 100,
            o_w: 50,
            o_h: 50,
            frame: 0,
            total_frames: 1,
            rot: 0.0,
        })
        .with(crate::components::Player {
            impulse: vector2d::Vector2D::new(0.0, 0.0),
            cur_speed: vector2d::Vector2D::new(0.0, 0.0),
        })
        .build();

    ecs.create_entity()
        .with(crate::components::Position {
            x: 400.0,
            y: 235.0,
            rot: 45.0,
        })
        .with(crate::components::Renderable {
            tex_name: String::from("img/asteroid.png"),
            i_w: 100,
            i_h: 100,
            o_w: 50,
            o_h: 50,
            frame: 0,
            total_frames: 1,
            rot: 0.0,
        })
        .with(crate::components::Asteroid {
            speed: 2.5,
            rot_speed: 0.5,
        })
        .build();
}
```

```
// ********** tutorial8 - texture_manager

// ********** tutorial8 - utils
// ***********************************************************************************
// ********** tutorial9 - main
use sdl2::event::Event;
use sdl2::keyboard::Keycode;
use sdl2::pixels::Color;
use sdl2::render::{TextureCreator, WindowCanvas};
use sdl2::video::WindowContext;

use sdl2::rect::Point;
use sdl2::rect::Rect;

use specs::{DispatcherBuilder, Join, World, WorldExt};

use std::collections::HashMap;
use std::path::Path;
use std::time::Duration;

pub mod asteroid;
pub mod components;
pub mod game;
pub mod texture_manager;
pub mod utils;

const GAME_WIDTH: u32 = 800;
const GAME_HEIGHT: u32 = 600;

fn render(
    canvas: &mut WindowCanvas,
    texture_manager: &mut texture_manager::TextureManager<WindowContext>,
    _texture_creator: &TextureCreator<WindowContext>,
    _font: &sdl2::ttf::Font,
    ecs: &World,
) -> Result<(), String> {
    //
    let color = Color::RGB(0, 0, 0);
    canvas.set_draw_color(color);
    canvas.clear();

    let positions = ecs.read_storage::<components::Position>();
    let renderables = ecs.read_storage::<components::Renderable>();

    for (renderable, pos) in (&renderables, &positions).join() {
        let src = Rect::new(0, 0, renderable.i_w, renderable.i_h);
        let x: i32 = pos.x as i32;
        let y: i32 = pos.y as i32;
        let dest = Rect::new(
            x - ((renderable.o_w / 2) as i32),
            y - ((renderable.o_h / 2) as i32),
            renderable.o_w,
            renderable.o_h,
        );

        let center = Point::new((renderable.o_w / 2) as i32, (renderable.o_h / 2) as i32);
        let texture = texture_manager.load(&renderable.tex_name)?;
        canvas.copy_ex(
            &texture,
            src,            //source rect
            dest,           // dest rect
            renderable.rot, // angle
            center,         // center
            false,          // flip horizontal
            false,          // flip vertical
        )?;
    }

    canvas.present();
    Ok(())
}

struct State {
    ecs: World,
}

fn main() -> Result<(), String> {
    println!("Starting Rusteroids");

    let sdl_context = sdl2::init()?;
    let video_subsystem = sdl_context.video()?;

    let window = video_subsystem
        .window("Rusteroids", GAME_WIDTH, GAME_HEIGHT)
        .position_centered()
        .build()
        .expect("could not initialize video subsystem");

    let mut canvas = window
        .into_canvas()
        .build()
        .expect("could not make a canvas");

    let texture_creator = canvas.texture_creator();
    let mut tex_man = texture_manager::TextureManager::new(&texture_creator);

    // Load the images before the main loop so we don't try and load during gameplay
    tex_man.load("img/space_ship.png")?;
    tex_man.load("img/asteroid.png")?;

    // Prepare fonts
    let ttf_context = sdl2::ttf::init().map_err(|e| e.to_string())?;
    let font_path: &Path = Path::new(&"fonts/OpenSans-Bold.ttf");
    let mut font = ttf_context.load_font(font_path, 128)?;
    font.set_style(sdl2::ttf::FontStyle::BOLD);

    let mut event_pump = sdl_context.event_pump()?;
    let mut key_manager: HashMap<String, bool> = HashMap::new();

    let mut gs = State { ecs: World::new() };
    gs.ecs.register::<components::Position>();
    gs.ecs.register::<components::Renderable>();
    gs.ecs.register::<components::Player>();
    gs.ecs.register::<components::Asteroid>();

    let mut dispatcher = DispatcherBuilder::new()
        .with(asteroid::AsteroidMover, "asteroid_mover", &[])
         .with(asteroid::AsteroidCollider, "asteroid_collider", &[])
        .build();
    game::load_world(&mut gs.ecs);

    'running: loop {
        // Handle events
        for event in event_pump.poll_iter() {
            match event {
                Event::Quit { .. } => {
                    break 'running;
                }
                Event::KeyDown {
                    keycode: Some(Keycode::Escape),
                    ..
                } => {
                    break 'running;
                }
                Event::KeyDown { keycode, .. } => match keycode {
                    None => {}
                    Some(key) => {
                        utils::key_down(&mut key_manager, key.to_string());
                    }
                },
                Event::KeyUp { keycode, .. } => match keycode {
                    None => {}
                    Some(key) => {
                        utils::key_up(&mut key_manager, key.to_string());
                    }
                },
                _ => {}
            }
        }

        game::update(&mut gs.ecs, &mut key_manager);
        dispatcher.dispatch(&gs.ecs);
         gs.ecs.maintain();
        render(&mut canvas, &mut tex_man, &texture_creator, &font, &gs.ecs)?;

        // Time management
        ::std::thread::sleep(Duration::new(0, 1_000_000_000u32 / 60));
    }

    Ok(())
}
```

```
// ********** tutorial9 - asteroid
use specs::prelude::Entities;
use specs::{Join, System, WriteStorage};

pub struct AsteroidMover;

use crate::components;

impl<'a> System<'a> for AsteroidMover {
    type SystemData = (
        WriteStorage<'a, components::Position>,
        WriteStorage<'a, components::Renderable>,
        WriteStorage<'a, components::Asteroid>,
    );

    fn run(&mut self, mut data: Self::SystemData) {
        for (pos, rend, asteroid) in (&mut data.0, &mut data.1, &data.2).join() {
            let radians = pos.rot.to_radians();

            pos.x += asteroid.speed * radians.sin();
            pos.y -= asteroid.speed * radians.cos();

            let half_width = (rend.o_w / 2) as u32;
            let half_height = (rend.o_h / 2) as u32;

            if pos.x > (crate::GAME_WIDTH - half_width).into() || pos.x < half_width.into() {
                pos.rot = 360.0 - pos.rot;
            } else if pos.y > (crate::GAME_HEIGHT - half_height).into()
                || pos.y < half_height.into()
            {
                if pos.rot > 180.0 {
                    pos.rot = 540.0 - pos.rot;
                } else {
                    pos.rot = 180.0 - pos.rot;
                }
            }

            rend.rot += asteroid.rot_speed;
            if rend.rot > 360.0 {
                rend.rot -= 360.0;
            }
            if rend.rot < 0.0 {
                rend.rot += 360.0;
            }
        }
    }
}

pub struct AsteroidCollider;

impl<'a> System<'a> for AsteroidCollider {
    type SystemData = (
        WriteStorage<'a, components::Position>,
        WriteStorage<'a, components::Renderable>,
        WriteStorage<'a, components::Player>,
        WriteStorage<'a, components::Asteroid>,
        Entities<'a>,
    );

    fn run(&mut self, data: Self::SystemData) {
        let (positions, rends, players, asteroids, entities) = data;

        for (player_pos, player_rend, _, entity) in (&positions, &rends, &players, &entities).join()
        {
            for (asteroid_pos, asteroid_rend, _) in (&positions, &rends, &asteroids).join() {
                let diff_x: f64 = (player_pos.x - asteroid_pos.x).abs();
                let diff_y: f64 = (player_pos.y - asteroid_pos.y).abs();
                let hyp: f64 = ((diff_x * diff_x) + (diff_y * diff_y)).sqrt();
                if hyp < (asteroid_rend.o_w + player_rend.o_w) as f64 / 2.0 {
                    println!("Player Died");
                    entities.delete(entity).ok();
                }
            }
        }
    }
}
```

// ********** tutorial9 - components

// ********** tutorial9 - game

```
use specs::{Builder, Join, World, WorldExt};
use std::collections::HashMap;
use vector2d::Vector2D;

const ROTATION_SPEED: f64 = 1.5;
const PLAYER_SPEED: f64 = 4.5;

pub fn update(ecs: &mut World, key_manager: &mut HashMap<String, bool>) {
    // Check status of game world
    let mut must_reload_world = false;
    {
        let players = ecs.read_storage::<crate::components::Player>();
        if players.join().count() < 1 {
            must_reload_world = true;
        }
    }

    if must_reload_world {
        // Remove all of the previous entities so we can start again
        ecs.delete_all();
        // Reset the world to first state
        load_world(ecs);
    }

    let mut positions = ecs.write_storage::<crate::components::Position>();
    let mut players = ecs.write_storage::<crate::components::Player>();
    let mut renderables = ecs.write_storage::<crate::components::Renderable>();

    for (player, pos, renderable) in (&mut players, &mut positions, &mut renderables).join() {
        if crate::utils::is_key_pressed(&key_manager, "D") {
            pos.rot += ROTATION_SPEED;
        }
        if crate::utils::is_key_pressed(&key_manager, "A") {
            pos.rot -= ROTATION_SPEED;
        }
        update_movement(pos, player);
        if crate::utils::is_key_pressed(&key_manager, "W") {
            let radians = pos.rot.to_radians();

            let move_x = PLAYER_SPEED * radians.sin();
            let move_y = PLAYER_SPEED * radians.cos();
            let move_vec = Vector2D::<f64>::new(move_x, move_y);

            player.impulse += move_vec;
        }

        if pos.rot > 360.0 {
            pos.rot -= 360.0;
        }
        if pos.rot < 0.0 {
            pos.rot += 360.0;
        }

        if pos.x > crate::GAME_WIDTH.into() {
            pos.x -= crate::GAME_WIDTH as f64;
        }
        if pos.x < 0.0 {
            pos.x += crate::GAME_WIDTH as f64;
        }
        if pos.y > crate::GAME_HEIGHT.into() {
            pos.y -= crate::GAME_HEIGHT as f64;
        }
        if pos.y < 0.0 {
            pos.y += crate::GAME_HEIGHT as f64;
        }

        // Update the graphic to reflect the rotation
        renderable.rot = pos.rot;
    }
}

const MAX_SPEED: f64 = 3.5;
const FRICTION: f64 = 0.99;

pub fn update_movement(
    pos: &mut crate::components::Position,
    player: &mut crate::components::Player,
) {
    player.cur_speed *= FRICTION;

    player.cur_speed += player.impulse;
    if player.cur_speed.length() > MAX_SPEED {
        player.cur_speed = player.cur_speed.normalise();
        player.cur_speed = player.cur_speed * MAX_SPEED;
    }

    pos.x += player.cur_speed.x;
    pos.y -= player.cur_speed.y;

    player.impulse = vector2d::Vector2D::new(0.0, 0.0);
}

pub fn load_world(ecs: &mut World) {
    ecs.create_entity()
        .with(crate::components::Position {
            x: 350.0,
            y: 250.0,
            rot: 0.0,
        })
        .with(crate::components::Renderable {
            tex_name: String::from("img/space_ship.png"),
            i_w: 100,
            i_h: 100,
            o_w: 50,
            o_h: 50,
            frame: 0,
            total_frames: 1,
            rot: 0.0,
        })
        .with(crate::components::Player {
            impulse: vector2d::Vector2D::new(0.0, 0.0),
            cur_speed: vector2d::Vector2D::new(0.0, 0.0),
        })
        .build();

    ecs.create_entity()
        .with(crate::components::Position {
            x: 400.0,
            y: 235.0,
            rot: 45.0,
        })
        .with(crate::components::Renderable {
            tex_name: String::from("img/asteroid.png"),
            i_w: 100,
            i_h: 100,
            o_w: 50,
            o_h: 50,
            frame: 0,
            total_frames: 1,
            rot: 0.0,
        })
        .with(crate::components::Asteroid {
            speed: 2.5,
            rot_speed: 0.5,
        })
        .build();
}
```

// ******** tutorial9 - texture_manager**

// ******** tutorial9 - utils**

```
// ***********************************************************************************
// ********** tutorial10 - main
use sdl2::event::Event;
use sdl2::keyboard::Keycode;
use sdl2::pixels::Color;
use sdl2::render::{TextureCreator, WindowCanvas};
use sdl2::video::WindowContext;

use sdl2::rect::Point;
use sdl2::rect::Rect;

use specs::{DispatcherBuilder, Join, World, WorldExt};

use std::collections::HashMap;
use std::path::Path;
use std::time::Duration;

pub mod asteroid;
pub mod components;
pub mod game;
pub mod missile;
pub mod texture_manager;
pub mod utils;

const GAME_WIDTH: u32 = 800;
const GAME_HEIGHT: u32 = 600;

fn render(
    canvas: &mut WindowCanvas,
    texture_manager: &mut texture_manager::TextureManager<WindowContext>,
    _texture_creator: &TextureCreator<WindowContext>,
    _font: &sdl2::ttf::Font,
    ecs: &World,
) -> Result<(), String> {
    //
    let color = Color::RGB(0, 0, 0);
    canvas.set_draw_color(color);
    canvas.clear();

    let positions = ecs.read_storage::<components::Position>();
    let renderables = ecs.read_storage::<components::Renderable>();

    for (renderable, pos) in (&renderables, &positions).join() {
        let src = Rect::new(0, 0, renderable.i_w, renderable.i_h);
        let x: i32 = pos.x as i32;
        let y: i32 = pos.y as i32;
        let dest = Rect::new(
            x - ((renderable.o_w / 2) as i32),
            y - ((renderable.o_h / 2) as i32),
            renderable.o_w,
            renderable.o_h,
        );

        let center = Point::new((renderable.o_w / 2) as i32, (renderable.o_h / 2) as i32);
        let texture = texture_manager.load(&renderable.tex_name)?;
        canvas.copy_ex(
            &texture,
            src,            //source rect
            dest,           // dest rect
            renderable.rot, // angle
            center,         // center
            false,          // flip horizontal
            false,          // flip vertical
        )?;
    }

    canvas.present();
    Ok(())
}

struct State {
    ecs: World,
}

fn main() -> Result<(), String> {
    println!("Starting Rusteroids");

    let sdl_context = sdl2::init()?;
    let video_subsystem = sdl_context.video()?;

    let window = video_subsystem
        .window("Rusteroids", GAME_WIDTH, GAME_HEIGHT)
        .position_centered()
        .build()
        .expect("could not initialize video subsystem");

    let mut canvas = window
        .into_canvas()
        .build()
        .expect("could not make a canvas");

    let texture_creator = canvas.texture_creator();
    let mut tex_man = texture_manager::TextureManager::new(&texture_creator);

    // Load the images before the main loop so we don't try and load during gameplay
    tex_man.load("img/space_ship.png")?;
    tex_man.load("img/asteroid.png")?;
     tex_man.load("img/missile.png")?;

    // Prepare fonts
    let ttf_context = sdl2::ttf::init().map_err(|e| e.to_string())?;
    let font_path: &Path = Path::new(&"fonts/OpenSans-Bold.ttf");
    let mut font = ttf_context.load_font(font_path, 128)?;
    font.set_style(sdl2::ttf::FontStyle::BOLD);

    let mut event_pump = sdl_context.event_pump()?;
    let mut key_manager: HashMap<String, bool> = HashMap::new();

    let mut gs = State { ecs: World::new() };
    gs.ecs.register::<components::Position>();
    gs.ecs.register::<components::Renderable>();
    gs.ecs.register::<components::Player>();
    gs.ecs.register::<components::Asteroid>();
     gs.ecs.register::<components::Missile>();
    let mut dispatcher = DispatcherBuilder::new()
        .with(asteroid::AsteroidMover, "asteroid_mover", &[])
        .with(asteroid::AsteroidCollider, "asteroid_collider", &[])
        .with(missile::MissileMover, "missile_mover", &[])
        .build();

    game::load_world(&mut gs.ecs);

    'running: loop {
        // Handle events
        for event in event_pump.poll_iter() {
            match event {
                Event::Quit { .. } => {
                    break 'running;
                }
                Event::KeyDown {
                    keycode: Some(Keycode::Escape),
                    ..
                } => {
                    break 'running;
                }
                Event::KeyDown {
                    keycode: Some(Keycode::Space),
                    ..
                } => {
                    utils::key_down(&mut key_manager, " ".to_string());
                }
                Event::KeyUp {
                    keycode: Some(Keycode::Space),
                    ..
                } => {
                    utils::key_up(&mut key_manager, " ".to_string());
                }
                Event::KeyDown { keycode, .. } => match keycode {
                    None => {}
                    Some(key) => {
                        utils::key_down(&mut key_manager, key.to_string());
                    }
                },
                Event::KeyUp { keycode, .. } => match keycode {
                    None => {}
                    Some(key) => {
                        utils::key_up(&mut key_manager, key.to_string());
                    }
                },
                _ => {}
            }
        }

        game::update(&mut gs.ecs, &mut key_manager);
        dispatcher.dispatch(&gs.ecs);
        gs.ecs.maintain();
        render(&mut canvas, &mut tex_man, &texture_creator, &font, &gs.ecs)?;

        // Time management
        ::std::thread::sleep(Duration::new(0, 1_000_000_000u32 / 60));
    }

    Ok(())
}
```

// ********** tutorial10 - asteroid

// ********** tutorial10 - components

```
use specs::prelude::*;
use specs_derive::Component;
use vector2d::Vector2D;

#[derive(Component)]
pub struct Position {
    pub x: f64,
    pub y: f64,
    pub rot: f64,
}

#[derive(Component)]
pub struct Renderable {
    pub tex_name: String,
    pub i_w: u32,
    pub i_h: u32,
    pub o_w: u32,
    pub o_h: u32,
    pub frame: u32,
    pub total_frames: u32,
    pub rot: f64,
}

#[derive(Component)]
pub struct Player {
    pub impulse: Vector2D<f64>,   // The next impulse to add to the speed
    pub cur_speed: Vector2D<f64>, // The current speed of the player
}

#[derive(Component)]
pub struct Asteroid {
    pub speed: f64,
    pub rot_speed: f64,
}

#[derive(Component)]
pub struct Missile {
    pub speed: f64,
}
```

// ********** tutorial10 - game

```
use specs::{Builder, Join, World, WorldExt};
use std::collections::HashMap;
use vector2d::Vector2D;

use crate::components;
use crate::utils;
```

```
const ROTATION_SPEED: f64 = 1.5;
const PLAYER_SPEED: f64 = 4.5;

pub fn update(ecs: &mut World, key_manager: &mut HashMap<String, bool>) {
    // Check status of game world
    let mut must_reload_world = false;
    {
        let players = ecs.read_storage::<crate::components::Player>();
        if players.join().count() < 1 {
            must_reload_world = true;
        }
    }

    if must_reload_world {
        // Remove all of the previous entities so we can start again
        ecs.delete_all();
        // Reset the world to first state
        load_world(ecs);
    }
    let mut player_pos = components::Position {
        x: 0.0,
        y: 0.0,
        rot: 0.0,
    };
    let mut must_fire_missile = false;

    {
        let mut positions = ecs.write_storage::<crate::components::Position>();
        let mut players = ecs.write_storage::<crate::components::Player>();
        let mut renderables = ecs.write_storage::<crate::components::Renderable>();

        for (player, pos, renderable) in (&mut players, &mut positions, &mut renderables).join() {
            if crate::utils::is_key_pressed(&key_manager, "D") {
                pos.rot += ROTATION_SPEED;
            }
            if crate::utils::is_key_pressed(&key_manager, "A") {
                pos.rot -= ROTATION_SPEED;
            }
            update_movement(pos, player);
            if crate::utils::is_key_pressed(&key_manager, "W") {
                let radians = pos.rot.to_radians();

                let move_x = PLAYER_SPEED * radians.sin();
                let move_y = PLAYER_SPEED * radians.cos();
                let move_vec = Vector2D::<f64>::new(move_x, move_y);

                player.impulse += move_vec;
            }

            if pos.rot > 360.0 {
                pos.rot -= 360.0;
            }
            if pos.rot < 0.0 {
                pos.rot += 360.0;
            }

            if pos.x > crate::GAME_WIDTH.into() {
                pos.x -= crate::GAME_WIDTH as f64;
            }
            if pos.x < 0.0 {
                pos.x += crate::GAME_WIDTH as f64;
            }
            if pos.y > crate::GAME_HEIGHT.into() {
                pos.y -= crate::GAME_HEIGHT as f64;
            }
            if pos.y < 0.0 {
                pos.y += crate::GAME_HEIGHT as f64;
            }

            if utils::is_key_pressed(&key_manager, " ") {
                utils::key_up(key_manager, " ".to_string());
                must_fire_missile = true;
                player_pos.x = pos.x;
                player_pos.y = pos.y;
                player_pos.rot = pos.rot;
            }

            // Update the graphic to reflect the rotation
            renderable.rot = pos.rot;
        }
    }

    if must_fire_missile {
        fire_missile(ecs, player_pos);
    }
}

const MAX_SPEED: f64 = 3.5;
const FRICTION: f64 = 0.99;

pub fn update_movement(
    pos: &mut crate::components::Position,
    player: &mut crate::components::Player,
) {
    player.cur_speed *= FRICTION;

    player.cur_speed += player.impulse;
    if player.cur_speed.length() > MAX_SPEED {
        player.cur_speed = player.cur_speed.normalise();
        player.cur_speed = player.cur_speed * MAX_SPEED;
    }

    pos.x += player.cur_speed.x;
    pos.y -= player.cur_speed.y;
    player.impulse = vector2d::Vector2D::new(0.0, 0.0);
}

pub fn load_world(ecs: &mut World) {
    ecs.create_entity()
        .with(crate::components::Position {
            x: 350.0,
            y: 250.0,
            rot: 0.0,
        })
        .with(crate::components::Renderable {
            tex_name: String::from("img/space_ship.png"),
            i_w: 100,
            i_h: 100,
            o_w: 50,
            o_h: 50,
            frame: 0,
            total_frames: 1,
            rot: 0.0,
        })
        .with(crate::components::Player {
            impulse: vector2d::Vector2D::new(0.0, 0.0),
            cur_speed: vector2d::Vector2D::new(0.0, 0.0),
        })
        .build();

    ecs.create_entity()
        .with(crate::components::Position {
            x: 400.0,
            y: 235.0,
            rot: 45.0,
        })
        .with(crate::components::Renderable {
            tex_name: String::from("img/asteroid.png"),
            i_w: 100,
            i_h: 100,
            o_w: 50,
            o_h: 50,
            frame: 0,
            total_frames: 1,
            rot: 0.0,
        })
        .with(crate::components::Asteroid {
            speed: 2.5,
            rot_speed: 0.5,
        })
        .build();
}

const MAX_MISSILES: usize = 3;

fn fire_missile(ecs: &mut World, position: components::Position) {
    {
        let missiles = ecs.read_storage::<components::Missile>();
        if missiles.count() > MAX_MISSILES - 1 {
            return;
        }
    }
    ecs.create_entity()
        .with(position)
        .with(crate::components::Renderable {
            tex_name: String::from("img/missile.png"),
            i_w: 50,
            i_h: 100,
            o_w: 10,
            o_h: 20,
            frame: 0,
            total_frames: 1,
            rot: 0.0,
        })
        .with(crate::components::Missile { speed: 5.0 })
        .build();
}
```

```
// ********** tutorial10 - missile
use specs::prelude::*;
use specs::{Entities, Join};

use crate::components;

pub struct MissileMover;

impl<'a> System<'a> for MissileMover {
    type SystemData = (
        WriteStorage<'a, components::Position>,
        WriteStorage<'a, components::Renderable>,
        ReadStorage<'a, components::Missile>,
        Entities<'a>,
    );

    fn run(&mut self, data: Self::SystemData) {
        let (mut positions, mut renderables, missiles, entities) = data;

        for (pos, rend, missile, entity) in
            (&mut positions, &mut renderables, &missiles, &entities).join()
        {
            let radian = pos.rot.to_radians();

            let move_x = missile.speed * radian.sin();
            let move_y = missile.speed * radian.cos();
            pos.x += move_x;
            pos.y -= move_y;
            if pos.x > crate::GAME_WIDTH.into()
                || pos.x < 0.0
                || pos.y > crate::GAME_HEIGHT.into()
                || pos.y < 0.0
            {
                entities.delete(entity).ok();
            }

            rend.rot = pos.rot;
        }
    }
}
```

```
// ********** tutorial10 - texture_manager
```

```
// ********** tutorial10 - utils
```

```rust
// ********************************************************************************
// ********** tutorial11 - main

// ********** tutorial11 - asteroid

// ********** tutorial11 - components

// ********** tutorial11 - game
use specs::{Builder, Join, World, WorldExt};
use std::collections::HashMap;
use vector2d::Vector2D;

use crate::components;
use crate::utils;

const ROTATION_SPEED: f64 = 1.5;
const PLAYER_SPEED: f64 = 4.5;

pub fn update(ecs: &mut World, key_manager: &mut HashMap<String, bool>) {
    // Check status of game world
    let mut must_reload_world = false;
    let mut current_player_position = components::Position {
        x: 0.0,
        y: 0.0,
        rot: 0.0,
    };
    {
        let players = ecs.read_storage::<components::Player>();
        let positions = ecs.read_storage::<components::Position>();

        for (pos, _player) in (&positions, &players).join() {
            current_player_position.x = pos.x;
            current_player_position.y = pos.y;
        }

        if players.join().count() < 1 {
            must_reload_world = true;
        }
    }

    if must_reload_world {
        // Remove all of the previous entities so we can start again
        ecs.delete_all();
        // Reset the world to first state
        load_world(ecs);
    }

    // Check if all asteroids are missing
    let mut must_create_asteroid = false;
    {
        let asteroids = ecs.read_storage::<components::Asteroid>();
        if asteroids.join().count() < 1 {
            must_create_asteroid = true;
        }
    }
    if must_create_asteroid {
        if current_player_position.x > (crate::GAME_WIDTH / 2).into()
            && current_player_position.y < (crate::GAME_HEIGHT / 2).into()
        {
            // Player top right
            current_player_position.x = crate::GAME_WIDTH as f64 / 4.0;
            current_player_position.y =
                crate::GAME_HEIGHT as f64 - (crate::GAME_HEIGHT as f64 / 4.0);
            current_player_position.rot = 225.0;
        } else if current_player_position.x < (crate::GAME_WIDTH / 2).into()
            && current_player_position.y < (crate::GAME_HEIGHT / 2).into()
        {
            // Player top left
            current_player_position.x = crate::GAME_WIDTH as f64 - (crate::GAME_WIDTH as f64 / 4.0);
            current_player_position.y =
                crate::GAME_HEIGHT as f64 - (crate::GAME_HEIGHT as f64 / 4.0);
            current_player_position.rot = 135.0;
        } else if current_player_position.x > (crate::GAME_WIDTH / 2).into()
            && current_player_position.y > (crate::GAME_HEIGHT / 2).into()
        {
            // Player bottom right
            current_player_position.x = crate::GAME_WIDTH as f64 / 4.0;
            current_player_position.y = crate::GAME_HEIGHT as f64 / 4.0;
            current_player_position.rot = 315.0;
        } else if current_player_position.x < (crate::GAME_WIDTH / 2).into()
            && current_player_position.y > (crate::GAME_HEIGHT / 2).into()
        {
            // Player bottom left
            current_player_position.x = crate::GAME_WIDTH as f64 - (crate::GAME_WIDTH as f64 / 4.0);
            current_player_position.y = crate::GAME_HEIGHT as f64 / 4.0;
            current_player_position.rot = 45.0;
        }
        create_asteroid(ecs, current_player_position);
    }

    let mut player_pos = components::Position {
        x: 0.0,
        y: 0.0,
        rot: 0.0,
    };
    let mut must_fire_missile = false;

    {
        let mut positions = ecs.write_storage::<components::Position>();
        let mut players = ecs.write_storage::<components::Player>();
        let mut renderables = ecs.write_storage::<components::Renderable>();
        for (player, pos, renderable) in (&mut players, &mut positions, &mut renderables).join() {
            if crate::utils::is_key_pressed(&key_manager, "D") {
                pos.rot += ROTATION_SPEED;
            }
            if crate::utils::is_key_pressed(&key_manager, "A") {
                pos.rot -= ROTATION_SPEED;
            }
            update_movement(pos, player);
            if crate::utils::is_key_pressed(&key_manager, "W") {
                let radians = pos.rot.to_radians();

                let move_x = PLAYER_SPEED * radians.sin();
                let move_y = PLAYER_SPEED * radians.cos();
                let move_vec = Vector2D::<f64>::new(move_x, move_y);

                player.impulse += move_vec;
            }

            if pos.rot > 360.0 {
                pos.rot -= 360.0;
            }
            if pos.rot < 0.0 {
                pos.rot += 360.0;
            }

            if pos.x > crate::GAME_WIDTH.into() {
                pos.x -= crate::GAME_WIDTH as f64;
            }
            if pos.x < 0.0 {
                pos.x += crate::GAME_WIDTH as f64;
            }
            if pos.y > crate::GAME_HEIGHT.into() {
                pos.y -= crate::GAME_HEIGHT as f64;
            }
            if pos.y < 0.0 {
                pos.y += crate::GAME_HEIGHT as f64;
            }

            if utils::is_key_pressed(&key_manager, " ") {
                utils::key_up(key_manager, " ".to_string());
                must_fire_missile = true;
                player_pos.x = pos.x;
                player_pos.y = pos.y;
                player_pos.rot = pos.rot;
            }

            // Update the graphic to reflect the rotation
            renderable.rot = pos.rot;
        }
    }

    if must_fire_missile {
        fire_missile(ecs, player_pos);
    }
}

const MAX_SPEED: f64 = 3.5;
const FRICTION: f64 = 0.99;

pub fn update_movement(pos: &mut components::Position, player: &mut components::Player) {
    player.cur_speed *= FRICTION;

    player.cur_speed += player.impulse;
    if player.cur_speed.length() > MAX_SPEED {
        player.cur_speed = player.cur_speed.normalise();
        player.cur_speed = player.cur_speed * MAX_SPEED;
    }

    pos.x += player.cur_speed.x;
    pos.y -= player.cur_speed.y;

    player.impulse = vector2d::Vector2D::new(0.0, 0.0);
}

pub fn load_world(ecs: &mut World) {
    ecs.create_entity()
        .with(components::Position {
            x: 350.0,
            y: 250.0,
            rot: 0.0,
        })
        .with(components::Renderable {
            tex_name: String::from("img/space_ship.png"),
            i_w: 100,
            i_h: 100,
            o_w: 50,
            o_h: 50,
            frame: 0,
            total_frames: 1,
            rot: 0.0,
        })
        .with(components::Player {
            impulse: vector2d::Vector2D::new(0.0, 0.0),
            cur_speed: vector2d::Vector2D::new(0.0, 0.0),
        })
        .build();
    create_asteroid(
        ecs,
        components::Position {
            x: 400.0,
            y: 235.0,
            rot: 45.0,
        },
    );
}

pub fn create_asteroid(ecs: &mut World, position: components::Position) {
    ecs.create_entity()
        .with(position)
        .with(components::Renderable {
            tex_name: String::from("img/asteroid.png"),
            i_w: 100,
            i_h: 100,
            o_w: 50,
            o_h: 50,
            frame: 0,
            total_frames: 1,
            rot: 0.0,
        })
        .with(crate::components::Asteroid {
            speed: 2.5,
            rot_speed: 0.5,
        })
        .build();
}

const MAX_MISSILES: usize = 3;

fn fire_missile(ecs: &mut World, position: components::Position) {
    {
        let missiles = ecs.read_storage::<components::Missile>();
        if missiles.count() > MAX_MISSILES - 1 {
            return;
        }
    }
    ecs.create_entity()
        .with(position)
        .with(components::Renderable {
            tex_name: String::from("img/missile.png"),
            i_w: 50,
            i_h: 100,
            o_w: 10,
            o_h: 20,
            frame: 0,
            total_frames: 1,
            rot: 0.0,
        })
        .with(components::Missile { speed: 5.0 })
        .build();
}
```

// ********** tutorial11 - missile

```
use specs::prelude::*;
use specs::{Entities, Join};

use crate::components;

pub struct MissileMover;

impl<'a> System<'a> for MissileMover {
    type SystemData = (
        WriteStorage<'a, components::Position>,
        WriteStorage<'a, components::Renderable>,
        ReadStorage<'a, components::Missile>,
        Entities<'a>,
    );

    fn run(&mut self, data: Self::SystemData) {
        let (mut positions, mut renderables, missiles, entities) = data;

        for (pos, rend, missile, entity) in
            (&mut positions, &mut renderables, &missiles, &entities).join()
        {
            let radian = pos.rot.to_radians();

            let move_x = missile.speed * radian.sin();
            let move_y = missile.speed * radian.cos();
            pos.x += move_x;
            pos.y -= move_y;
            if pos.x > crate::GAME_WIDTH.into()
                || pos.x < 0.0
                || pos.y > crate::GAME_HEIGHT.into()
                || pos.y < 0.0
            {
                entities.delete(entity).ok();
            }

            rend.rot = pos.rot;
        }
    }
}

pub struct MissileStriker;

impl<'a> System<'a> for MissileStriker {
    type SystemData = (
        WriteStorage<'a, components::Position>,
        WriteStorage<'a, components::Renderable>,
        WriteStorage<'a, components::Missile>,
        WriteStorage<'a, components::Asteroid>,
        WriteStorage<'a, components::Player>,
        Entities<'a>,
    );

    fn run(&mut self, data: Self::SystemData) {
        let (positions, rends, missiles, asteroids, _players, entities) = &data;

        for (missile_pos, _, _, missile_entity) in (positions, rends, missiles, entities).join() {
            for (asteroid_pos, asteroid_rend, _, asteroid_entity) in
                (positions, rends, asteroids, entities).join()
            {
                let diff_x: f64 = (missile_pos.x - asteroid_pos.x).abs();
                let diff_y: f64 = (missile_pos.y - asteroid_pos.y).abs();
                let hyp: f64 = ((diff_x * diff_x) + (diff_y * diff_y)).sqrt();
                if hyp < asteroid_rend.o_w as f64 / 2.0 {
                    entities.delete(missile_entity).ok();
                    entities.delete(asteroid_entity).ok();
                }
            }
        }
    }
}
```

// ********** tutorial11 - texture_manager

```
// ********** tutorial11 - utils.rs
// ***********************************************************************************
// ********** tutorial12 - main

// ********** tutorial12 - asteroid

// ********** tutorial12 - components
use specs::prelude::*;
use specs_derive::Component;
use vector2d::Vector2D;

#[derive(Component)]
pub struct Position {
    pub x: f64,
    pub y: f64,
    pub rot: f64,
}

#[derive(Component)]
pub struct Renderable {
    pub tex_name: String,
    pub i_w: u32,
    pub i_h: u32,
    pub o_w: u32,
    pub o_h: u32,
    pub frame: u32,
    pub total_frames: u32,
    pub rot: f64,
}

#[derive(Component)]
pub struct Player {
    pub impulse: Vector2D<f64>,   // The next impulse to add to the speed
    pub cur_speed: Vector2D<f64>, // The current speed of the player
}

#[derive(Component)]
pub struct Asteroid {
    pub speed: f64,
    pub rot_speed: f64,
}

#[derive(Component)]
pub struct Missile {
    pub speed: f64,
}

pub struct PendingAsteroid {
    pub x: f64,
    pub y: f64,
    pub rot: f64,
    pub size: u32,
}

// ********** tutorial12 - game
use specs::{Builder, Join, World, WorldExt};
use std::collections::HashMap;
use vector2d::Vector2D;

use crate::components;
use crate::utils;

const ROTATION_SPEED: f64 = 1.5;
const PLAYER_SPEED: f64 = 4.5;

pub fn update(ecs: &mut World, key_manager: &mut HashMap<String, bool>) {
    // Check status of game world
    let mut must_reload_world = false;
    let mut current_player_position = components::Position {
        x: 0.0,
        y: 0.0,
        rot: 0.0,
    };
    {
        let players = ecs.read_storage::<components::Player>();
        let positions = ecs.read_storage::<components::Position>();

        for (pos, _player) in (&positions, &players).join() {
            current_player_position.x = pos.x;
            current_player_position.y = pos.y;
        }

        if players.join().count() < 1 {
            must_reload_world = true;
        }
    }

    if must_reload_world {
        // Remove all of the previous entities so we can start again
        ecs.delete_all();
        // Reset the world to first state
        load_world(ecs);
    }

    // Check if all asteroids are missing
    let mut must_create_asteroid = false;
    {
        let asteroids = ecs.read_storage::<components::Asteroid>();
        if asteroids.join().count() < 1 {
            must_create_asteroid = true;
        }
    }
    if must_create_asteroid {
        if current_player_position.x > (crate::GAME_WIDTH / 2).into()
            && current_player_position.y < (crate::GAME_HEIGHT / 2).into()
        {
            // Player top right
            current_player_position.x = crate::GAME_WIDTH as f64 / 4.0;
            current_player_position.y =
                crate::GAME_HEIGHT as f64 - (crate::GAME_HEIGHT as f64 / 4.0);
            current_player_position.rot = 225.0;
        } else if current_player_position.x < (crate::GAME_WIDTH / 2).into()
            && current_player_position.y < (crate::GAME_HEIGHT / 2).into()
        {
            // Player top left
            current_player_position.x = crate::GAME_WIDTH as f64 - (crate::GAME_WIDTH as f64 / 4.0);
            current_player_position.y =
                crate::GAME_HEIGHT as f64 - (crate::GAME_HEIGHT as f64 / 4.0);
            current_player_position.rot = 135.0;
        } else if current_player_position.x > (crate::GAME_WIDTH / 2).into()
            && current_player_position.y > (crate::GAME_HEIGHT / 2).into()
        {
            // Player bottom right
            current_player_position.x = crate::GAME_WIDTH as f64 / 4.0;
            current_player_position.y = crate::GAME_HEIGHT as f64 / 4.0;
            current_player_position.rot = 315.0;
        } else if current_player_position.x < (crate::GAME_WIDTH / 2).into()
            && current_player_position.y > (crate::GAME_HEIGHT / 2).into()
        {
            // Player bottom left
            current_player_position.x = crate::GAME_WIDTH as f64 - (crate::GAME_WIDTH as f64 / 4.0);
            current_player_position.y = crate::GAME_HEIGHT as f64 / 4.0;
            current_player_position.rot = 45.0;
        }
        create_asteroid(ecs, current_player_position, 100);
    }

    let mut player_pos = components::Position {
        x: 0.0,
        y: 0.0,
        rot: 0.0,
    };
    let mut must_fire_missile = false;

    {
        let mut positions = ecs.write_storage::<components::Position>();
        let mut players = ecs.write_storage::<components::Player>();
        let mut renderables = ecs.write_storage::<components::Renderable>();

        for (player, pos, renderable) in (&mut players, &mut positions, &mut renderables).join() {
            if crate::utils::is_key_pressed(&key_manager, "D") {
                pos.rot += ROTATION_SPEED;
            }
            if crate::utils::is_key_pressed(&key_manager, "A") {
                pos.rot -= ROTATION_SPEED;
            }
            update_movement(pos, player);
            if crate::utils::is_key_pressed(&key_manager, "W") {
                let radians = pos.rot.to_radians();

                let move_x = PLAYER_SPEED * radians.sin();
                let move_y = PLAYER_SPEED * radians.cos();
                let move_vec = Vector2D::<f64>::new(move_x, move_y);

                player.impulse += move_vec;
            }

            if pos.rot > 360.0 {
                pos.rot -= 360.0;
            }
            if pos.rot < 0.0 {
                pos.rot += 360.0;
            }

            if pos.x > crate::GAME_WIDTH.into() {
                pos.x -= crate::GAME_WIDTH as f64;
            }
            if pos.x < 0.0 {
                pos.x += crate::GAME_WIDTH as f64;
            }
            if pos.y > crate::GAME_HEIGHT.into() {
                pos.y -= crate::GAME_HEIGHT as f64;
            }
            if pos.y < 0.0 {
                pos.y += crate::GAME_HEIGHT as f64;
            }

            if utils::is_key_pressed(&key_manager, " ") {
                utils::key_up(key_manager, " ".to_string());
                must_fire_missile = true;
                player_pos.x = pos.x;
                player_pos.y = pos.y;
                player_pos.rot = pos.rot;
            }

            // Update the graphic to reflect the rotation
            renderable.rot = pos.rot;
        }
    }

    if must_fire_missile {
        fire_missile(ecs, player_pos);
    }
}

const MAX_SPEED: f64 = 3.5;
const FRICTION: f64 = 0.99;

pub fn update_movement(pos: &mut components::Position, player: &mut components::Player) {
    player.cur_speed *= FRICTION;

    player.cur_speed += player.impulse;
    if player.cur_speed.length() > MAX_SPEED {
        player.cur_speed = player.cur_speed.normalise();
        player.cur_speed = player.cur_speed * MAX_SPEED;
    }

    pos.x += player.cur_speed.x;
    pos.y -= player.cur_speed.y;

    player.impulse = vector2d::Vector2D::new(0.0, 0.0);
}

pub fn load_world(ecs: &mut World) {
    ecs.create_entity()
        .with(components::Position {
            x: 350.0,
            y: 250.0,
            rot: 0.0,
        })
        .with(components::Renderable {
            tex_name: String::from("img/space_ship.png"),
            i_w: 100,
            i_h: 100,
            o_w: 50,
            o_h: 50,
            frame: 0,
            total_frames: 1,
            rot: 0.0,
        })
        .with(components::Player {
            impulse: vector2d::Vector2D::new(0.0, 0.0),
            cur_speed: vector2d::Vector2D::new(0.0, 0.0),
        })
        .build();
    create_asteroid(
        ecs,
        components::Position {
            x: 400.0,
            y: 235.0,
            rot: 45.0,
        },
         50,
    );
}

pub fn create_asteroid(ecs: &mut World, position: components::Position, asteroid_size: u32) {
    ecs.create_entity()
        .with(position)
        .with(components::Renderable {
            tex_name: String::from("img/asteroid.png"),
            i_w: 100,
            i_h: 100,
              o_w: asteroid_size,
              o_h: asteroid_size,
            frame: 0,
            total_frames: 1,
            rot: 0.0,
        })
        .with(crate::components::Asteroid {
            speed: 2.5,
            rot_speed: 0.5,
        })
        .build();
}

const MAX_MISSILES: usize = 3;

fn fire_missile(ecs: &mut World, position: components::Position) {
    {
        let missiles = ecs.read_storage::<components::Missile>();
        if missiles.count() > MAX_MISSILES - 1 {
            return;
        }
    }
    ecs.create_entity()
        .with(position)
        .with(components::Renderable {
            tex_name: String::from("img/missile.png"),
            i_w: 50,
            i_h: 100,
            o_w: 10,
            o_h: 20,
            frame: 0,
            total_frames: 1,
            rot: 0.0,
        })
        .with(components::Missile { speed: 5.0 })
        .build();
}
```

// ********** tutorial12 - missile

```
use specs::prelude::*;
use specs::{Entities, Join};

use crate::components;

pub struct MissileMover;

impl<'a> System<'a> for MissileMover {
    type SystemData = (
        WriteStorage<'a, components::Position>,
        WriteStorage<'a, components::Renderable>,
        ReadStorage<'a, components::Missile>,
        Entities<'a>,
    );

    fn run(&mut self, data: Self::SystemData) {
        let (mut positions, mut renderables, missiles, entities) = data;

        for (pos, rend, missile, entity) in
            (&mut positions, &mut renderables, &missiles, &entities).join()
        {
            let radian = pos.rot.to_radians();

            let move_x = missile.speed * radian.sin();
            let move_y = missile.speed * radian.cos();
            pos.x += move_x;
            pos.y -= move_y;
            if pos.x > crate::GAME_WIDTH.into()
                || pos.x < 0.0
                || pos.y > crate::GAME_HEIGHT.into()
                || pos.y < 0.0
            {
                entities.delete(entity).ok();
            }

            rend.rot = pos.rot;
        }
    }
}

pub struct MissileStriker;

impl<'a> System<'a> for MissileStriker {
    type SystemData = (
        WriteStorage<'a, components::Position>,
        WriteStorage<'a, components::Renderable>,
        WriteStorage<'a, components::Missile>,
        WriteStorage<'a, components::Asteroid>,
        WriteStorage<'a, components::Player>,
        Entities<'a>,
    );

    fn run(&mut self, data: Self::SystemData) {
        let (positions, rends, missiles, asteroids, _players, entities) = &data;
        let mut asteroid_creation = Vec::<components::PendingAsteroid>::new();

        for (missile_pos, _, _, missile_entity) in (positions, rends, missiles, entities).join() {
            for (asteroid_pos, asteroid_rend, _, asteroid_entity) in
                (positions, rends, asteroids, entities).join()
            {
                let diff_x: f64 = (missile_pos.x - asteroid_pos.x).abs();
                let diff_y: f64 = (missile_pos.y - asteroid_pos.y).abs();
                let hyp: f64 = ((diff_x * diff_x) + (diff_y * diff_y)).sqrt();
                if hyp < asteroid_rend.o_w as f64 / 2.0 {
                    entities.delete(missile_entity).ok();
                    entities.delete(asteroid_entity).ok();
                    let new_size = asteroid_rend.o_w / 2;
                    if new_size >= 25 {
                        asteroid_creation.push(components::PendingAsteroid {
                            x: asteroid_pos.x,
                            y: asteroid_pos.y,
                            rot: asteroid_pos.rot - 90.0,
                            size: new_size,
                        });
                        asteroid_creation.push(components::PendingAsteroid {
                            x: asteroid_pos.x,
                            y: asteroid_pos.y,
                            rot: asteroid_pos.rot + 90.0,
                            size: new_size,
                        });
                    }
                }
            }
        }

        let (mut positions, mut rends, _, mut asteroids, _, entities) = data;
        for new_asteroid in asteroid_creation {
            let new_ast = entities.create();
            positions
                .insert(
                    new_ast,
                    components::Position {
                        x: new_asteroid.x,
                        y: new_asteroid.y,
                        rot: new_asteroid.rot,
                    },
                )
                .ok();
            asteroids
                .insert(
                    new_ast,
                    components::Asteroid {
                        speed: 2.5,
                        rot_speed: 0.5,
                    },
                )
                .ok();
            rends
                .insert(
                    new_ast,
                    components::Renderable {
                        tex_name: String::from("img/asteroid.png"),
                        i_w: 100,
                        i_h: 100,
                        o_w: new_asteroid.size,
                        o_h: new_asteroid.size,
                        frame: 0,
                        total_frames: 1,
                        rot: 0.0,
                    },
                )
                .ok();
        }
    }
}

// ********** tutorial12 - texture_manager

// ********** tutorial12 - utils
// ********************************************************************************
// ********** tutorial13 - main
use sdl2::event::Event;
use sdl2::keyboard::Keycode;
use sdl2::pixels::Color;
use sdl2::render::{TextureCreator, WindowCanvas};
use sdl2::video::WindowContext;

use sdl2::rect::Point;
use sdl2::rect::Rect;

use specs::{DispatcherBuilder, Join, World, WorldExt};

use std::collections::HashMap;
use std::path::Path;
use std::time::Duration;

pub mod asteroid;
pub mod components;
pub mod game;
pub mod missile;
pub mod texture_manager;
pub mod utils;

const GAME_WIDTH: u32 = 800;
const GAME_HEIGHT: u32 = 600;

fn render(
    canvas: &mut WindowCanvas,
    texture_manager: &mut texture_manager::TextureManager<WindowContext>,
    texture_creator: &TextureCreator<WindowContext>,
    font: &sdl2::ttf::Font,
    ecs: &World,
) -> Result<(), String> {
    //
    let color = Color::RGB(0, 0, 0);
    canvas.set_draw_color(color);
    canvas.clear();

    let positions = ecs.read_storage::<components::Position>();
    let renderables = ecs.read_storage::<components::Renderable>();

    for (renderable, pos) in (&renderables, &positions).join() {
        let src = Rect::new(0, 0, renderable.i_w, renderable.i_h);
        let x: i32 = pos.x as i32;
        let y: i32 = pos.y as i32;
        let dest = Rect::new(
            x - ((renderable.o_w / 2) as i32),
            y - ((renderable.o_h / 2) as i32),
            renderable.o_w,
            renderable.o_h,
        );

        let center = Point::new((renderable.o_w / 2) as i32, (renderable.o_h / 2) as i32);
        let texture = texture_manager.load(&renderable.tex_name)?;
        canvas.copy_ex(
            &texture,
            src,            //source rect
            dest,           // dest rect
            renderable.rot, // angle
            center,         // center
            false,          // flip horizontal
            false,          // flip vertical
        )?;
    }

    let gamedatas = ecs.read_storage::<components::GameData>();
    for gamedata in (gamedatas).join() {
        // Show Score
        let score: String = "Score: ".to_string() + &gamedata.score.to_string();
        let surface = font
            .render(&score)
            .blended(Color::RGBA(255, 0, 0, 128))
            .map_err(|e| e.to_string())?;
        let texture = texture_creator
            .create_texture_from_surface(&surface)
            .map_err(|e| e.to_string())?;

        let target = Rect::new(10 as i32, 0 as i32, 100 as u32, 50 as u32);
        canvas.copy(&texture, None, Some(target))?;
    }

    canvas.present();
    Ok(())
}

struct State {
    ecs: World,
}

fn main() -> Result<(), String> {
    println!("Starting Rusteroids");

    let sdl_context = sdl2::init()?;
    let video_subsystem = sdl_context.video()?;

    let window = video_subsystem
        .window("Rusteroids", GAME_WIDTH, GAME_HEIGHT)
        .position_centered()
        .build()
        .expect("could not initialize video subsystem");

    let mut canvas = window
        .into_canvas()
        .build()
        .expect("could not make a canvas");

    let texture_creator = canvas.texture_creator();
    let mut tex_man = texture_manager::TextureManager::new(&texture_creator);

    // Load the images before the main loop so we don't try and load during gameplay
    tex_man.load("img/space_ship.png")?;
    tex_man.load("img/asteroid.png")?;
    tex_man.load("img/missile.png")?;

    // Prepare fonts
    let ttf_context = sdl2::ttf::init().map_err(|e| e.to_string())?;
    let font_path: &Path = Path::new(&"fonts/OpenSans-Bold.ttf");
    let mut font = ttf_context.load_font(font_path, 128)?;
    font.set_style(sdl2::ttf::FontStyle::BOLD);

    let mut event_pump = sdl_context.event_pump()?;
    let mut key_manager: HashMap<String, bool> = HashMap::new();

    let mut gs = State { ecs: World::new() };
    gs.ecs.register::<components::Position>();
    gs.ecs.register::<components::Renderable>();
    gs.ecs.register::<components::Player>();
    gs.ecs.register::<components::Asteroid>();
    gs.ecs.register::<components::Missile>();
    gs.ecs.register::<components::GameData>();

    let mut dispatcher = DispatcherBuilder::new()
        .with(asteroid::AsteroidMover, "asteroid_mover", &[])
        .with(asteroid::AsteroidCollider, "asteroid_collider", &[])
        .with(missile::MissileMover, "missile_mover", &[])
        .with(missile::MissileStriker, "missile_striker", &[])
        .build();

    game::load_world(&mut gs.ecs);

    'running: loop {
        // Handle events
        for event in event_pump.poll_iter() {
            match event {
                Event::Quit { .. } => {
                    break 'running;
                }
                Event::KeyDown {
                    keycode: Some(Keycode::Escape),
                    ..
                } => {
                    break 'running;
                }
                Event::KeyDown {
                    keycode: Some(Keycode::Space),
                    ..
                } => {
                    utils::key_down(&mut key_manager, " ".to_string());
                }
                Event::KeyUp {
                    keycode: Some(Keycode::Space),
                    ..
                } => {
                    utils::key_up(&mut key_manager, " ".to_string());
                }
                Event::KeyDown { keycode, .. } => match keycode {
                    None => {}
                    Some(key) => {
                        utils::key_down(&mut key_manager, key.to_string());
                    }
                },
                Event::KeyUp { keycode, .. } => match keycode {
                    None => {}
                    Some(key) => {
                        utils::key_up(&mut key_manager, key.to_string());
                    }
                },
                _ => {}
            }
        }

        game::update(&mut gs.ecs, &mut key_manager);
        dispatcher.dispatch(&gs.ecs);
        gs.ecs.maintain();
        render(&mut canvas, &mut tex_man, &texture_creator, &font, &gs.ecs)?;

        // Time management
        ::std::thread::sleep(Duration::new(0, 1_000_000_000u32 / 60));
    }

    Ok(())
}
```

// ******** tutorial13 - asteroid**

// ******** tutorial13 - components**

```
use specs::prelude::*;
use specs_derive::Component;
use vector2d::Vector2D;

#[derive(Component)]
pub struct Position {
    pub x: f64,
    pub y: f64,
    pub rot: f64,
}

#[derive(Component)]
pub struct Renderable {
    pub tex_name: String,
    pub i_w: u32,
    pub i_h: u32,
    pub o_w: u32,
    pub o_h: u32,
    pub frame: u32,
    pub total_frames: u32,
    pub rot: f64,
}

#[derive(Component)]
pub struct Player {
    pub impulse: Vector2D<f64>,   // The next impulse to add to the speed
    pub cur_speed: Vector2D<f64>, // The current speed of the player
}

#[derive(Component)]
pub struct Asteroid {
    pub speed: f64,
    pub rot_speed: f64,
}

#[derive(Component)]
pub struct Missile {
    pub speed: f64,
}

pub struct PendingAsteroid {
    pub x: f64,
    pub y: f64,
    pub rot: f64,
    pub size: u32,
}
```

**```
#[derive(Component)]
pub struct GameData {
    pub score: u32,
}
```**

## // ********** tutorial13 - game

```
use specs::{Builder, Join, World, WorldExt};
use std::collections::HashMap;
use vector2d::Vector2D;

use crate::components;
use crate::utils;

const ROTATION_SPEED: f64 = 1.5;
const PLAYER_SPEED: f64 = 4.5;

pub fn update(ecs: &mut World, key_manager: &mut HashMap<String, bool>) {
    // Check status of game world
    let mut must_reload_world = false;
    let mut current_player_position = components::Position {
        x: 0.0,
        y: 0.0,
        rot: 0.0,
    };
    {
        let players = ecs.read_storage::<components::Player>();
        let positions = ecs.read_storage::<components::Position>();

        for (pos, _player) in (&positions, &players).join() {
            current_player_position.x = pos.x;
            current_player_position.y = pos.y;
        }

        if players.join().count() < 1 {
            must_reload_world = true;
        }
    }

    if must_reload_world {
        // Remove all of the previous entities so we can start again
        ecs.delete_all();
        // Reset the world to first state
        load_world(ecs);
    }

    // Check if all asteroids are missing
    let mut must_create_asteroid = false;
    {
        let asteroids = ecs.read_storage::<components::Asteroid>();
        if asteroids.join().count() < 1 {
            must_create_asteroid = true;
        }
    }
    if must_create_asteroid {
        if current_player_position.x > (crate::GAME_WIDTH / 2).into()
            && current_player_position.y < (crate::GAME_HEIGHT / 2).into()
        {
            // Player top right
            current_player_position.x = crate::GAME_WIDTH as f64 / 4.0;
            current_player_position.y =
                crate::GAME_HEIGHT as f64 - (crate::GAME_HEIGHT as f64 / 4.0);
            current_player_position.rot = 225.0;
        } else if current_player_position.x < (crate::GAME_WIDTH / 2).into()
            && current_player_position.y < (crate::GAME_HEIGHT / 2).into()
        {
            // Player top left
            current_player_position.x = crate::GAME_WIDTH as f64 - (crate::GAME_WIDTH as f64 / 4.0);
            current_player_position.y =
                crate::GAME_HEIGHT as f64 - (crate::GAME_HEIGHT as f64 / 4.0);
            current_player_position.rot = 135.0;
        } else if current_player_position.x > (crate::GAME_WIDTH / 2).into()
            && current_player_position.y > (crate::GAME_HEIGHT / 2).into()
        {
            // Player bottom right
            current_player_position.x = crate::GAME_WIDTH as f64 / 4.0;
            current_player_position.y = crate::GAME_HEIGHT as f64 / 4.0;
            current_player_position.rot = 315.0;
        } else if current_player_position.x < (crate::GAME_WIDTH / 2).into()
            && current_player_position.y > (crate::GAME_HEIGHT / 2).into()
        {
            // Player bottom left
            current_player_position.x = crate::GAME_WIDTH as f64 - (crate::GAME_WIDTH as f64 / 4.0);
            current_player_position.y = crate::GAME_HEIGHT as f64 / 4.0;
            current_player_position.rot = 45.0;
        }
        create_asteroid(ecs, current_player_position, 100);
    }

    let mut player_pos = components::Position {
        x: 0.0,
        y: 0.0,
        rot: 0.0,
    };
    let mut must_fire_missile = false;

    {
        let mut positions = ecs.write_storage::<components::Position>();
        let mut players = ecs.write_storage::<components::Player>();
        let mut renderables = ecs.write_storage::<components::Renderable>();
```
```

```
        for (player, pos, renderable) in (&mut players, &mut positions, &mut renderables).join() {
            if crate::utils::is_key_pressed(&key_manager, "D") {
                pos.rot += ROTATION_SPEED;
            }
            if crate::utils::is_key_pressed(&key_manager, "A") {
                pos.rot -= ROTATION_SPEED;
            }
            update_movement(pos, player);
            if crate::utils::is_key_pressed(&key_manager, "W") {
                let radians = pos.rot.to_radians();

                let move_x = PLAYER_SPEED * radians.sin();
                let move_y = PLAYER_SPEED * radians.cos();
                let move_vec = Vector2D::<f64>::new(move_x, move_y);

                player.impulse += move_vec;
            }

            if pos.rot > 360.0 {
                pos.rot -= 360.0;
            }
            if pos.rot < 0.0 {
                pos.rot += 360.0;
            }

            if pos.x > crate::GAME_WIDTH.into() {
                pos.x -= crate::GAME_WIDTH as f64;
            }
            if pos.x < 0.0 {
                pos.x += crate::GAME_WIDTH as f64;
            }
            if pos.y > crate::GAME_HEIGHT.into() {
                pos.y -= crate::GAME_HEIGHT as f64;
            }
            if pos.y < 0.0 {
                pos.y += crate::GAME_HEIGHT as f64;
            }

            if utils::is_key_pressed(&key_manager, " ") {
                utils::key_up(key_manager, " ".to_string());
                must_fire_missile = true;
                player_pos.x = pos.x;
                player_pos.y = pos.y;
                player_pos.rot = pos.rot;
            }

            // Update the graphic to reflect the rotation
            renderable.rot = pos.rot;
        }
    }

    if must_fire_missile {
        fire_missile(ecs, player_pos);
    }
}

const MAX_SPEED: f64 = 3.5;
const FRICTION: f64 = 0.99;

pub fn update_movement(pos: &mut components::Position, player: &mut components::Player) {
    player.cur_speed *= FRICTION;

    player.cur_speed += player.impulse;
    if player.cur_speed.length() > MAX_SPEED {
        player.cur_speed = player.cur_speed.normalise();
        player.cur_speed = player.cur_speed * MAX_SPEED;
    }

    pos.x += player.cur_speed.x;
    pos.y -= player.cur_speed.y;

    player.impulse = vector2d::Vector2D::new(0.0, 0.0);
}

pub fn load_world(ecs: &mut World) {
    ecs.create_entity()
        .with(components::Position {
            x: 350.0,
            y: 250.0,
            rot: 0.0,
        })
        .with(components::Renderable {
            tex_name: String::from("img/space_ship.png"),
            i_w: 100,
            i_h: 100,
            o_w: 50,
            o_h: 50,
            frame: 0,
            total_frames: 1,
            rot: 0.0,
        })
        .with(components::Player {
            impulse: vector2d::Vector2D::new(0.0, 0.0),
            cur_speed: vector2d::Vector2D::new(0.0, 0.0),
        })
        .build();

    create_asteroid(
        ecs,
        components::Position {
            x: 400.0,
            y: 235.0,
            rot: 45.0,
        },
        50,
    );

    ecs.create_entity()
        .with(components::GameData { score: 0 })
        .build();
}

pub fn create_asteroid(ecs: &mut World, position: components::Position, asteroid_size: u32) {
    ecs.create_entity()
        .with(position)
        .with(components::Renderable {
            tex_name: String::from("img/asteroid.png"),
            i_w: 100,
            i_h: 100,
            o_w: asteroid_size,
            o_h: asteroid_size,
            frame: 0,
            total_frames: 1,
            rot: 0.0,
        })
        .with(crate::components::Asteroid {
            speed: 2.5,
            rot_speed: 0.5,
        })
        .build();
}

const MAX_MISSILES: usize = 3;

fn fire_missile(ecs: &mut World, position: components::Position) {
    {
        let missiles = ecs.read_storage::<components::Missile>();
        if missiles.count() > MAX_MISSILES - 1 {
            return;
        }
    }
    ecs.create_entity()
        .with(position)
        .with(components::Renderable {
            tex_name: String::from("img/missile.png"),
            i_w: 50,
            i_h: 100,
            o_w: 10,
            o_h: 20,
            frame: 0,
            total_frames: 1,
            rot: 0.0,
        })
        .with(components::Missile { speed: 5.0 })
        .build();
}
```

// ******** tutorial13 - missile**

```
use specs::prelude::*;
use specs::{Entities, Join};

use crate::components;

pub struct MissileMover;

impl<'a> System<'a> for MissileMover {
    type SystemData = (
        WriteStorage<'a, components::Position>,
        WriteStorage<'a, components::Renderable>,
        ReadStorage<'a, components::Missile>,
        Entities<'a>,
    );

    fn run(&mut self, data: Self::SystemData) {
        let (mut positions, mut renderables, missiles, entities) = data;

        for (pos, rend, missile, entity) in
            (&mut positions, &mut renderables, &missiles, &entities).join()
        {
            let radian = pos.rot.to_radians();

            let move_x = missile.speed * radian.sin();
            let move_y = missile.speed * radian.cos();
            pos.x += move_x;
            pos.y -= move_y;
            if pos.x > crate::GAME_WIDTH.into()
                || pos.x < 0.0
                || pos.y > crate::GAME_HEIGHT.into()
                || pos.y < 0.0
            {
                entities.delete(entity).ok();
            }

            rend.rot = pos.rot;
        }
    }
}

pub struct MissileStriker;

impl<'a> System<'a> for MissileStriker {
    type SystemData = (
        WriteStorage<'a, components::Position>,
        WriteStorage<'a, components::Renderable>,
        WriteStorage<'a, components::Missile>,
        WriteStorage<'a, components::Asteroid>,
        WriteStorage<'a, components::Player>,
        WriteStorage<'a, components::GameData>,
        Entities<'a>,
    );

    fn run(&mut self, data: Self::SystemData) {
        let (positions, rends, missiles, asteroids, _players, _, entities) = &data;
        let mut asteroid_creation = Vec::<components::PendingAsteroid>::new();
        let mut score: u32 = 0;

        for (missile_pos, _, _, missile_entity) in (positions, rends, missiles, entities).join() {
            for (asteroid_pos, asteroid_rend, _, asteroid_entity) in
                (positions, rends, asteroids, entities).join()
            {
                let diff_x: f64 = (missile_pos.x - asteroid_pos.x).abs();
                let diff_y: f64 = (missile_pos.y - asteroid_pos.y).abs();
                let hyp: f64 = ((diff_x * diff_x) + (diff_y * diff_y)).sqrt();
                if hyp < asteroid_rend.o_w as f64 / 2.0 {
                    score += 10;
                    entities.delete(missile_entity).ok();
                    entities.delete(asteroid_entity).ok();
                    let new_size = asteroid_rend.o_w / 2;
                    if new_size >= 25 {
                        asteroid_creation.push(components::PendingAsteroid {
                            x: asteroid_pos.x,
                            y: asteroid_pos.y,
                            rot: asteroid_pos.rot - 90.0,
                            size: new_size,
                        });
                        asteroid_creation.push(components::PendingAsteroid {
                            x: asteroid_pos.x,
                            y: asteroid_pos.y,
                            rot: asteroid_pos.rot + 90.0,
                            size: new_size,
                        });
                    }
                }
            }
        }

            let (mut positions, mut rends, _, mut asteroids, _, _, entities) = data;
        for new_asteroid in asteroid_creation {
            let new_ast = entities.create();
            positions
                .insert(
                    new_ast,
                    components::Position {
                        x: new_asteroid.x,
                        y: new_asteroid.y,
                        rot: new_asteroid.rot,
                    },
                )
                .ok();
            asteroids
                .insert(
                    new_ast,
                    components::Asteroid {
                        speed: 2.5,
                        rot_speed: 0.5,
                    },
                )
                .ok();
            rends
                .insert(
                    new_ast,
                    components::Renderable {
                        tex_name: String::from("img/asteroid.png"),
                        i_w: 100,
                        i_h: 100,
                        o_w: new_asteroid.size,
                        o_h: new_asteroid.size,
                        frame: 0,
                        total_frames: 1,
                        rot: 0.0,
                    },
                )
                .ok();
        }

            let (_, _, _, _, _, mut gamedatas, _) = data;
            for gamedata in (&mut gamedatas).join() {
                gamedata.score += score;
            }
    }
}

// ********** tutorial13 - texture_manager

// ********** tutorial13 - utils
// *************************************************************************************
// ********** tutorial14 - main

// ********** tutorial14 - asteroids

// ********** tutorial14 - components
use specs::prelude::*;
use specs_derive::Component;
use vector2d::Vector2D;

#[derive(Component)]
pub struct Position {
    pub x: f64,
    pub y: f64,
    pub rot: f64,
}

#[derive(Component)]
pub struct Renderable {
    pub tex_name: String,
    pub i_w: u32,
    pub i_h: u32,
    pub o_w: u32,
    pub o_h: u32,
    pub frame: u32,
    pub total_frames: u32,
    pub rot: f64,
}

#[derive(Component)]
pub struct Player {
    pub impulse: Vector2D<f64>,   // The next impulse to add to the speed
    pub cur_speed: Vector2D<f64>, // The current speed of the player
}

#[derive(Component)]
pub struct Asteroid {
    pub speed: f64,
    pub rot_speed: f64,
}

#[derive(Component)]
pub struct Missile {
    pub speed: f64,
}

pub struct PendingAsteroid {
    pub x: f64,
    pub y: f64,
    pub rot: f64,
    pub size: u32,
}

#[derive(Component)]
pub struct GameData {
    pub score: u32,
    pub level: u32,
}


// ********** tutorial14 - game
use rand::Rng;
use specs::{Builder, Join, World, WorldExt};
use std::collections::HashMap;
use vector2d::Vector2D;

use crate::components;
use crate::utils;

const ROTATION_SPEED: f64 = 1.5;
const PLAYER_SPEED: f64 = 4.5;

pub fn update(ecs: &mut World, key_manager: &mut HashMap<String, bool>) {
    // Check status of game world
    let mut must_reload_world = false;
    let mut current_player_position = components::Position {
        x: 0.0,
        y: 0.0,
        rot: 0.0,
    };
    {
        let players = ecs.read_storage::<components::Player>();
        let positions = ecs.read_storage::<components::Position>();

        for (pos, _player) in (&positions, &players).join() {
            current_player_position.x = pos.x;
            current_player_position.y = pos.y;
        }

        if players.join().count() < 1 {
            must_reload_world = true;
        }
    }

    if must_reload_world {
        // Remove all of the previous entities so we can start again
        ecs.delete_all();
        // Reset the world to first state
        load_world(ecs);
    }

    // Check if all asteroids are missing
    let mut must_create_asteroid = false;
    let mut number_asteroids: u32 = 0;
    {
        let asteroids = ecs.read_storage::<components::Asteroid>();
        if asteroids.join().count() < 1 {
            must_create_asteroid = true;

            let mut gamedatas = ecs.write_storage::<components::GameData>();
            for gamedata in (&mut gamedatas).join() {
                gamedata.level += 1;
                number_asteroids = (gamedata.level / 3) + 1;
            }
        }
    }

    if must_create_asteroid {
        let mut asteroid_count: u32 = 0;
        while asteroid_count < number_asteroids {
            let mut rng = rand::thread_rng();
            let next_x = rng.gen_range(50.0..(crate::GAME_WIDTH as f64 - 50.0));
            let next_y = rng.gen_range(50.0..(crate::GAME_HEIGHT as f64 - 50.0));
            let next_rot = rng.gen_range(0.0..360.0);

            let diff_x = (current_player_position.x - next_x).abs();
            let diff_y = (current_player_position.y - next_y).abs();
            if ((diff_x * diff_x) + (diff_y * diff_y)).sqrt() < 150.0 {
                // We are too close to the player
                continue;
            }
            asteroid_count += 1;
            let new_asteroid = components::Position {
                x: next_x,
                y: next_y,
                rot: next_rot,
            };
            create_asteroid(ecs, new_asteroid, 100);
        }
    }

    let mut player_pos = components::Position {
        x: 0.0,
        y: 0.0,
        rot: 0.0,
    };
    let mut must_fire_missile = false;

    {
        let mut positions = ecs.write_storage::<components::Position>();
        let mut players = ecs.write_storage::<components::Player>();
        let mut renderables = ecs.write_storage::<components::Renderable>();

        for (player, pos, renderable) in (&mut players, &mut positions, &mut renderables).join() {
            if crate::utils::is_key_pressed(&key_manager, "D") {
                pos.rot += ROTATION_SPEED;
            }
            if crate::utils::is_key_pressed(&key_manager, "A") {
                pos.rot -= ROTATION_SPEED;
            }
            update_movement(pos, player);
            if crate::utils::is_key_pressed(&key_manager, "W") {
                let radians = pos.rot.to_radians();

                let move_x = PLAYER_SPEED * radians.sin();
                let move_y = PLAYER_SPEED * radians.cos();
                let move_vec = Vector2D::<f64>::new(move_x, move_y);

                player.impulse += move_vec;
            }

            if pos.rot > 360.0 {
                pos.rot -= 360.0;
            }
            if pos.rot < 0.0 {
                pos.rot += 360.0;
            }

            if pos.x > crate::GAME_WIDTH.into() {
                pos.x -= crate::GAME_WIDTH as f64;
            }
            if pos.x < 0.0 {
                pos.x += crate::GAME_WIDTH as f64;
            }
            if pos.y > crate::GAME_HEIGHT.into() {
                pos.y -= crate::GAME_HEIGHT as f64;
            }
            if pos.y < 0.0 {
                pos.y += crate::GAME_HEIGHT as f64;
            }

            if utils::is_key_pressed(&key_manager, " ") {
                utils::key_up(key_manager, " ".to_string());
                must_fire_missile = true;
                player_pos.x = pos.x;
                player_pos.y = pos.y;
                player_pos.rot = pos.rot;
            }

            // Update the graphic to reflect the rotation
            renderable.rot = pos.rot;
        }
    }

    if must_fire_missile {
        fire_missile(ecs, player_pos);
    }
}

const MAX_SPEED: f64 = 3.5;
const FRICTION: f64 = 0.99;

pub fn update_movement(pos: &mut components::Position, player: &mut components::Player) {
    player.cur_speed *= FRICTION;

    player.cur_speed += player.impulse;
    if player.cur_speed.length() > MAX_SPEED {
        player.cur_speed = player.cur_speed.normalise();
        player.cur_speed = player.cur_speed * MAX_SPEED;
    }

    pos.x += player.cur_speed.x;
    pos.y -= player.cur_speed.y;

    player.impulse = vector2d::Vector2D::new(0.0, 0.0);
```

```
}

pub fn load_world(ecs: &mut World) {
    ecs.create_entity()
        .with(components::Position {
            x: 350.0,
            y: 250.0,
            rot: 0.0,
        })
        .with(components::Renderable {
            tex_name: String::from("img/space_ship.png"),
            i_w: 100,
            i_h: 100,
            o_w: 50,
            o_h: 50,
            frame: 0,
            total_frames: 1,
            rot: 0.0,
        })
        .with(components::Player {
            impulse: vector2d::Vector2D::new(0.0, 0.0),
            cur_speed: vector2d::Vector2D::new(0.0, 0.0),
        })
        .build();

    create_asteroid(
        ecs,
        components::Position {
            x: 400.0,
            y: 235.0,
            rot: 45.0,
        },
        50,
    );

    ecs.create_entity()
         .with(components::GameData { score: 0, level: 1 })
        .build();
}

pub fn create_asteroid(ecs: &mut World, position: components::Position, asteroid_size: u32) {
    ecs.create_entity()
        .with(position)
        .with(components::Renderable {
            tex_name: String::from("img/asteroid.png"),
            i_w: 100,
            i_h: 100,
            o_w: asteroid_size,
            o_h: asteroid_size,
            frame: 0,
            total_frames: 1,
            rot: 0.0,
        })
        .with(crate::components::Asteroid {
            speed: 2.5,
            rot_speed: 0.5,
        })
        .build();
}

const MAX_MISSILES: usize = 5;

fn fire_missile(ecs: &mut World, position: components::Position) {
    {
        let missiles = ecs.read_storage::<components::Missile>();
        if missiles.count() > MAX_MISSILES - 1 {
            return;
        }
    }
    ecs.create_entity()
        .with(position)
        .with(components::Renderable {
            tex_name: String::from("img/missile.png"),
            i_w: 50,
            i_h: 100,
            o_w: 10,
            o_h: 20,
            frame: 0,
            total_frames: 1,
            rot: 0.0,
        })
        .with(components::Missile { speed: 5.0 })
        .build();
}
```

// ******** tutorial14 - missile**

// ******** tutorial14 - texture_manager**

// ******** tutorial14 - utils**

```
// ********************************************************************************
// ********** tutorial15 - main
use sdl2::event::Event;
use sdl2::keyboard::Keycode;
use sdl2::pixels::Color;
use sdl2::render::{TextureCreator, WindowCanvas};
use sdl2::video::WindowContext;

use sdl2::rect::Point;
use sdl2::rect::Rect;

use specs::{DispatcherBuilder, Join, World, WorldExt};

use std::collections::HashMap;
use std::path::Path;
use std::time::Duration;

pub mod asteroid;
pub mod components;
pub mod game;
pub mod missile;
pub mod texture_manager;
pub mod utils;

const GAME_WIDTH: u32 = 800;
const GAME_HEIGHT: u32 = 600;

fn render(
    canvas: &mut WindowCanvas,
    texture_manager: &mut texture_manager::TextureManager<WindowContext>,
    texture_creator: &TextureCreator<WindowContext>,
    font: &sdl2::ttf::Font,
    ecs: &World,
) -> Result<(), String> {
    //
    let color = Color::RGB(0, 0, 0);
    canvas.set_draw_color(color);
    canvas.clear();

    let positions = ecs.read_storage::<components::Position>();
    {
        canvas.set_draw_color(Color::RGBA(255, 255, 255, 128));

        let stars = ecs.read_storage::<components::Star>();
        for (pos, star) in (&positions, &stars).join() {
            canvas.fill_rect(Rect::new(pos.x as i32, pos.y as i32, star.size, star.size))?;
        }
    }

    let renderables = ecs.read_storage::<components::Renderable>();

    for (renderable, pos) in (&renderables, &positions).join() {
        let src = Rect::new(0, 0, renderable.i_w, renderable.i_h);
        let x: i32 = pos.x as i32;
        let y: i32 = pos.y as i32;
        let dest = Rect::new(
            x - ((renderable.o_w / 2) as i32),
            y - ((renderable.o_h / 2) as i32),
            renderable.o_w,
            renderable.o_h,
        );

        let center = Point::new((renderable.o_w / 2) as i32, (renderable.o_h / 2) as i32);
        let texture = texture_manager.load(&renderable.tex_name)?;
        canvas.copy_ex(
            &texture,
            src,            //source rect
            dest,           // dest rect
            renderable.rot, // angle
            center,         // center
            false,          // flip horizontal
            false,          // flip vertical
        )?;
    }

    let gamedatas = ecs.read_storage::<components::GameData>();
    for gamedata in (gamedatas).join() {
        // Show Score
        let score: String = "Score: ".to_string() + &gamedata.score.to_string();
        let surface = font
            .render(&score)
            .blended(Color::RGBA(255, 0, 0, 128))
            .map_err(|e| e.to_string())?;
        let texture = texture_creator
            .create_texture_from_surface(&surface)
            .map_err(|e| e.to_string())?;

        let target = Rect::new(10 as i32, 0 as i32, 100 as u32, 50 as u32);
        canvas.copy(&texture, None, Some(target))?;
    }

    canvas.present();
    Ok(())
}

struct State {
    ecs: World,
}

fn main() -> Result<(), String> {
    println!("Starting Rusteroids");

    let sdl_context = sdl2::init()?;
    let video_subsystem = sdl_context.video()?;

    let window = video_subsystem
        .window("Rusteroids", GAME_WIDTH, GAME_HEIGHT)
        .position_centered()
        .build()
        .expect("could not initialize video subsystem");

    let mut canvas = window
        .into_canvas()
        .build()
        .expect("could not make a canvas");
    let texture_creator = canvas.texture_creator();
    let mut tex_man = texture_manager::TextureManager::new(&texture_creator);

    // Load the images before the main loop so we don't try and load during gameplay
    tex_man.load("img/space_ship.png")?;
    tex_man.load("img/asteroid.png")?;
    tex_man.load("img/missile.png")?;

    // Prepare fonts
    let ttf_context = sdl2::ttf::init().map_err(|e| e.to_string())?;
    let font_path: &Path = Path::new(&"fonts/OpenSans-Bold.ttf");
    let mut font = ttf_context.load_font(font_path, 128)?;
    font.set_style(sdl2::ttf::FontStyle::BOLD);

    let mut event_pump = sdl_context.event_pump()?;
    let mut key_manager: HashMap<String, bool> = HashMap::new();

    let mut gs = State { ecs: World::new() };
    gs.ecs.register::<components::Position>();
    gs.ecs.register::<components::Renderable>();
    gs.ecs.register::<components::Player>();
    gs.ecs.register::<components::Asteroid>();
    gs.ecs.register::<components::Missile>();
    gs.ecs.register::<components::GameData>();
    gs.ecs.register::<components::Star>();

    let mut dispatcher = DispatcherBuilder::new()
        .with(asteroid::AsteroidMover, "asteroid_mover", &[])
        .with(asteroid::AsteroidCollider, "asteroid_collider", &[])
        .with(missile::MissileMover, "missile_mover", &[])
        .with(missile::MissileStriker, "missile_striker", &[])
        .build();

    game::load_world(&mut gs.ecs);

    'running: loop {
        // Handle events
        for event in event_pump.poll_iter() {
            match event {
                Event::Quit { .. } => {
                    break 'running;
                }
                Event::KeyDown {
                    keycode: Some(Keycode::Escape),
                    ..
                } => {
                    break 'running;
                }
                Event::KeyDown {
                    keycode: Some(Keycode::Space),
                    ..
                } => {
                    utils::key_down(&mut key_manager, " ".to_string());
                }
                Event::KeyUp {
                    keycode: Some(Keycode::Space),
                    ..
                } => {
                    utils::key_up(&mut key_manager, " ".to_string());
                }
                Event::KeyDown { keycode, .. } => match keycode {
                    None => {}
                    Some(key) => {
                        utils::key_down(&mut key_manager, key.to_string());
                    }
                },
                Event::KeyUp { keycode, .. } => match keycode {
                    None => {}
                    Some(key) => {
                        utils::key_up(&mut key_manager, key.to_string());
                    }
                },
                _ => {}
            }
        }

        game::update(&mut gs.ecs, &mut key_manager);
        dispatcher.dispatch(&gs.ecs);
        gs.ecs.maintain();
        render(&mut canvas, &mut tex_man, &texture_creator, &font, &gs.ecs)?;

        // Time management
        ::std::thread::sleep(Duration::new(0, 1_000_000_000u32 / 60));
    }

    Ok(())
}
```

// ********** tutorial15 - asteroid

// ********** tutorial15 - components.rs

```
use specs::prelude::*;
use specs_derive::Component;
use vector2d::Vector2D;

#[derive(Component)]
pub struct Position {
    pub x: f64,
    pub y: f64,
    pub rot: f64,
}

#[derive(Component)]
pub struct Renderable {
    pub tex_name: String,
    pub i_w: u32,
    pub i_h: u32,
    pub o_w: u32,
    pub o_h: u32,
    pub frame: u32,
    pub total_frames: u32,
    pub rot: f64,
}
```

```
#[derive(Component)]
pub struct Player {
    pub impulse: Vector2D<f64>,   // The next impulse to add to the speed
    pub cur_speed: Vector2D<f64>, // The current speed of the player
}

#[derive(Component)]
pub struct Asteroid {
    pub speed: f64,
    pub rot_speed: f64,
}

#[derive(Component)]
pub struct Missile {
    pub speed: f64,
}

pub struct PendingAsteroid {
    pub x: f64,
    pub y: f64,
    pub rot: f64,
    pub size: u32,
}

#[derive(Component)]
pub struct GameData {
    pub score: u32,
    pub level: u32,
}
```

```
#[derive(Component)]
pub struct Star {
    pub size: u32,
}
```

// ******** tutorial15 - game.rs**

```
use rand::Rng;
use specs::{Builder, Join, World, WorldExt};
use std::collections::HashMap;
use vector2d::Vector2D;

use crate::components;
use crate::utils;

const ROTATION_SPEED: f64 = 1.5;
const PLAYER_SPEED: f64 = 4.5;

pub fn update(ecs: &mut World, key_manager: &mut HashMap<String, bool>) {
    // Check status of game world
    let mut must_reload_world = false;
    let mut current_player_position = components::Position {
        x: 0.0,
        y: 0.0,
        rot: 0.0,
    };
    {
        let players = ecs.read_storage::<components::Player>();
        let positions = ecs.read_storage::<components::Position>();

        for (pos, _player) in (&positions, &players).join() {
            current_player_position.x = pos.x;
            current_player_position.y = pos.y;
        }

        if players.join().count() < 1 {
            must_reload_world = true;
        }
    }

    if must_reload_world {
        // Remove all of the previous entities so we can start again
        ecs.delete_all();
        // Reset the world to first state
        load_world(ecs);
    }

    // Check if all asteroids are missing
    let mut must_create_asteroid = false;
    let mut number_asteroids: u32 = 0;
    {
        let asteroids = ecs.read_storage::<components::Asteroid>();
        if asteroids.join().count() < 1 {
            must_create_asteroid = true;

            let mut gamedatas = ecs.write_storage::<components::GameData>();
            for gamedata in (&mut gamedatas).join() {
                gamedata.level += 1;
                number_asteroids = (gamedata.level / 3) + 1;
            }
        }
    }

    if must_create_asteroid {
        let mut asteroid_count: u32 = 0;
        while asteroid_count < number_asteroids {
            let mut rng = rand::thread_rng();
            let next_x = rng.gen_range(50.0..(crate::GAME_WIDTH as f64 - 50.0));
            let next_y = rng.gen_range(50.0..(crate::GAME_HEIGHT as f64 - 50.0));
            let next_rot = rng.gen_range(0.0..360.0);

            let diff_x = (current_player_position.x - next_x).abs();
            let diff_y = (current_player_position.y - next_y).abs();
            if ((diff_x * diff_x) + (diff_y * diff_y)).sqrt() < 150.0 {
                // We are too close to the player
                continue;
            }
            asteroid_count += 1;
            let new_asteroid = components::Position {
                x: next_x,
                y: next_y,
                rot: next_rot,
            };
            create_asteroid(ecs, new_asteroid, 100);
        }
    }

    let mut player_pos = components::Position {
        x: 0.0,
        y: 0.0,
        rot: 0.0,
    };
    let mut must_fire_missile = false;

    {
        let mut positions = ecs.write_storage::<components::Position>();
        let mut players = ecs.write_storage::<components::Player>();
        let mut renderables = ecs.write_storage::<components::Renderable>();

        for (player, pos, renderable) in (&mut players, &mut positions, &mut renderables).join() {
            if crate::utils::is_key_pressed(&key_manager, "D") {
                pos.rot += ROTATION_SPEED;
            }
            if crate::utils::is_key_pressed(&key_manager, "A") {
                pos.rot -= ROTATION_SPEED;
            }
            update_movement(pos, player);
            if crate::utils::is_key_pressed(&key_manager, "W") {
                let radians = pos.rot.to_radians();

                let move_x = PLAYER_SPEED * radians.sin();
                let move_y = PLAYER_SPEED * radians.cos();
                let move_vec = Vector2D::<f64>::new(move_x, move_y);

                player.impulse += move_vec;
            }

            if pos.rot > 360.0 {
                pos.rot -= 360.0;
            }
            if pos.rot < 0.0 {
                pos.rot += 360.0;
            }

            if pos.x > crate::GAME_WIDTH.into() {
                pos.x -= crate::GAME_WIDTH as f64;
            }
            if pos.x < 0.0 {
                pos.x += crate::GAME_WIDTH as f64;
            }
            if pos.y > crate::GAME_HEIGHT.into() {
                pos.y -= crate::GAME_HEIGHT as f64;
            }
            if pos.y < 0.0 {
                pos.y += crate::GAME_HEIGHT as f64;
            }

            if utils::is_key_pressed(&key_manager, " ") {
                utils::key_up(key_manager, " ".to_string());
                must_fire_missile = true;
                player_pos.x = pos.x;
                player_pos.y = pos.y;
                player_pos.rot = pos.rot;
            }

            // Update the graphic to reflect the rotation
            renderable.rot = pos.rot;
        }
    }

    if must_fire_missile {
        fire_missile(ecs, player_pos);
    }
}

const MAX_SPEED: f64 = 3.5;
const FRICTION: f64 = 0.99;

pub fn update_movement(pos: &mut components::Position, player: &mut components::Player) {
    player.cur_speed *= FRICTION;

    player.cur_speed += player.impulse;
    if player.cur_speed.length() > MAX_SPEED {
        player.cur_speed = player.cur_speed.normalise();
        player.cur_speed = player.cur_speed * MAX_SPEED;
    }

    pos.x += player.cur_speed.x;
    pos.y -= player.cur_speed.y;

    player.impulse = vector2d::Vector2D::new(0.0, 0.0);
}
```

`pub const NUMBER_OF_STARS: u32 = 45;`

```
pub fn load_world(ecs: &mut World) {
    ecs.create_entity()
        .with(components::Position {
            x: 350.0,
            y: 250.0,
            rot: 0.0,
        })
        .with(components::Renderable {
            tex_name: String::from("img/space_ship.png"),
            i_w: 100,
            i_h: 100,
            o_w: 50,
            o_h: 50,
            frame: 0,
            total_frames: 1,
            rot: 0.0,
        })
        .with(components::Player {
            impulse: vector2d::Vector2D::new(0.0, 0.0),
            cur_speed: vector2d::Vector2D::new(0.0, 0.0),
        })
        .build();

    create_asteroid(
        ecs,
        components::Position {
            x: 400.0,
            y: 235.0,
            rot: 45.0,
        },
        50,
    );

    ecs.create_entity()
        .with(components::GameData { score: 0, level: 1 })
        .build();

    for _ in 0..=NUMBER_OF_STARS {
        let mut rng = rand::thread_rng();
        let star_x = rng.gen_range(0.0..crate::GAME_WIDTH as f64);
        let star_y = rng.gen_range(0.0..crate::GAME_HEIGHT as f64);
        let size = rng.gen_range(1..=3);

        ecs.create_entity()
            .with(components::Position {
                x: star_x,
                y: star_y,
                rot: 0.0,
            })
            .with(components::Star { size: size })
            .build();
    }
}

pub fn create_asteroid(ecs: &mut World, position: components::Position, asteroid_size: u32) {
    ecs.create_entity()
        .with(position)
        .with(components::Renderable {
            tex_name: String::from("img/asteroid.png"),
            i_w: 100,
            i_h: 100,
            o_w: asteroid_size,
            o_h: asteroid_size,
            frame: 0,
            total_frames: 1,
            rot: 0.0,
        })
        .with(crate::components::Asteroid {
            speed: 2.5,
            rot_speed: 0.5,
        })
        .build();
}

const MAX_MISSILES: usize = 5;

fn fire_missile(ecs: &mut World, position: components::Position) {
    {
        let missiles = ecs.read_storage::<components::Missile>();
        if missiles.count() > MAX_MISSILES - 1 {
            return;
        }
    }
    ecs.create_entity()
        .with(position)
        .with(components::Renderable {
            tex_name: String::from("img/missile.png"),
            i_w: 50,
            i_h: 100,
            o_w: 10,
            o_h: 20,
            frame: 0,
            total_frames: 1,
            rot: 0.0,
        })
        .with(components::Missile { speed: 5.0 })
        .build();
}
```

// ********** tutorial15 - missile

// ********** tutorial15 - texture_manager

// ********** tutorial15 - utils

```
// *********************************************************************************
// ********** tutorial16 - main
use sdl2::event::Event;
use sdl2::keyboard::Keycode;
use sdl2::pixels::Color;
use sdl2::render::{TextureCreator, WindowCanvas};
use sdl2::video::WindowContext;

use sdl2::rect::Point;
use sdl2::rect::Rect;

use specs::{DispatcherBuilder, Join, World, WorldExt};

use std::collections::HashMap;
use std::path::Path;
use std::sync::Mutex;
use std::time::Duration;

use once_cell::sync::Lazy;

pub mod asteroid;
pub mod components;
pub mod game;
pub mod missile;
pub mod texture_manager;
pub mod utils;

const GAME_WIDTH: u32 = 800;
const GAME_HEIGHT: u32 = 600;

fn render(
    canvas: &mut WindowCanvas,
    texture_manager: &mut texture_manager::TextureManager<WindowContext>,
    texture_creator: &TextureCreator<WindowContext>,
    font: &sdl2::ttf::Font,
    ecs: &World,
) -> Result<(), String> {
    //
    let color = Color::RGB(0, 0, 0);
    canvas.set_draw_color(color);
    canvas.clear();

    let positions = ecs.read_storage::<components::Position>();
    {
        canvas.set_draw_color(Color::RGBA(255, 255, 255, 128));

        let stars = ecs.read_storage::<components::Star>();
        for (pos, star) in (&positions, &stars).join() {
            canvas.fill_rect(Rect::new(pos.x as i32, pos.y as i32, star.size, star.size))?;
        }
    }

    let renderables = ecs.read_storage::<components::Renderable>();

    for (renderable, pos) in (&renderables, &positions).join() {
        let src = Rect::new(0, 0, renderable.i_w, renderable.i_h);
        let x: i32 = pos.x as i32;
        let y: i32 = pos.y as i32;
        let dest = Rect::new(
            x - ((renderable.o_w / 2) as i32),
            y - ((renderable.o_h / 2) as i32),
            renderable.o_w,
            renderable.o_h,
        );

        let center = Point::new((renderable.o_w / 2) as i32, (renderable.o_h / 2) as i32);
        let texture = texture_manager.load(&renderable.tex_name)?;
        canvas.copy_ex(
            &texture,
            src,            //source rect
            dest,           // dest rect
            renderable.rot, // angle
            center,         // center
            false,          // flip horizontal
            false,          // flip vertical
        )?;
    }

    let gamedatas = ecs.read_storage::<components::GameData>();
    for gamedata in (gamedatas).join() {
        // Show Score
        let score: String = "Score: ".to_string() + &gamedata.score.to_string();
        let surface = font
            .render(&score)
            .blended(Color::RGBA(255, 0, 0, 128))
            .map_err(|e| e.to_string())?;
        let texture = texture_creator
            .create_texture_from_surface(&surface)
            .map_err(|e| e.to_string())?;

        let target = Rect::new(10 as i32, 0 as i32, 100 as u32, 50 as u32);
        canvas.copy(&texture, None, Some(target))?;

        // Show Level
        let level: String = "Level: ".to_string() + &gamedata.level.to_string();
        let surface = font
            .render(&level)
            .blended(Color::RGBA(255, 0, 0, 128))
            .map_err(|e| e.to_string())?;
        let texture = texture_creator
            .create_texture_from_surface(&surface)
            .map_err(|e| e.to_string())?;

        let target = Rect::new(
            (crate::GAME_WIDTH - 110) as i32,
            0 as i32,
            100 as u32,
            50 as u32,
        );
        canvas.copy(&texture, None, Some(target))?;

        // High Score
        let high_score_value = &GAMESTATE.lock().unwrap().high_score;
        let high_score: String = "High Score: ".to_string() + &high_score_value.to_string();
        let surface = font
            .render(&high_score)
            .blended(Color::RGBA(255, 0, 0, 128))
            .map_err(|e| e.to_string())?;
        let texture = texture_creator
            .create_texture_from_surface(&surface)
            .map_err(|e| e.to_string())?;

        let target = Rect::new(
            10 as i32,
            (crate::GAME_HEIGHT - 60) as i32,
            100 as u32,
            50 as u32,
        );
        canvas.copy(&texture, None, Some(target))?;
    }

    canvas.present();
    Ok(())
}

struct State {
    ecs: World,
}

pub struct GameState {
    high_score: u32,
}

static GAMESTATE: Lazy<Mutex<GameState>> = Lazy::new(|| Mutex::new(GameState { high_score: 0 }));

fn main() -> Result<(), String> {
    println!("Starting Rusteroids");

    let sdl_context = sdl2::init()?;
    let video_subsystem = sdl_context.video()?;

    let window = video_subsystem
        .window("Rusteroids", GAME_WIDTH, GAME_HEIGHT)
        .position_centered()
        .build()
        .expect("could not initialize video subsystem");

    let mut canvas = window
        .into_canvas()
        .build()
        .expect("could not make a canvas");

    let texture_creator = canvas.texture_creator();
    let mut tex_man = texture_manager::TextureManager::new(&texture_creator);

    // Load the images before the main loop so we don't try and load during gameplay
    tex_man.load("img/space_ship.png")?;
    tex_man.load("img/asteroid.png")?;
    tex_man.load("img/missile.png")?;

    // Prepare fonts
    let ttf_context = sdl2::ttf::init().map_err(|e| e.to_string())?;
    let font_path: &Path = Path::new(&"fonts/OpenSans-Bold.ttf");
    let mut font = ttf_context.load_font(font_path, 128)?;
    font.set_style(sdl2::ttf::FontStyle::BOLD);

    let mut event_pump = sdl_context.event_pump()?;
    let mut key_manager: HashMap<String, bool> = HashMap::new();

    let mut gs = State { ecs: World::new() };
    gs.ecs.register::<components::Position>();
    gs.ecs.register::<components::Renderable>();
    gs.ecs.register::<components::Player>();
    gs.ecs.register::<components::Asteroid>();
    gs.ecs.register::<components::Missile>();
    gs.ecs.register::<components::GameData>();
    gs.ecs.register::<components::Star>();

    let mut dispatcher = DispatcherBuilder::new()
        .with(asteroid::AsteroidMover, "asteroid_mover", &[])
        .with(asteroid::AsteroidCollider, "asteroid_collider", &[])
        .with(missile::MissileMover, "missile_mover", &[])
        .with(missile::MissileStriker, "missile_striker", &[])
        .build();

    game::load_world(&mut gs.ecs);

    'running: loop {
        // Handle events
        for event in event_pump.poll_iter() {
            match event {
                Event::Quit { .. } => {
                    break 'running;
                }
                Event::KeyDown {
                    keycode: Some(Keycode::Escape),
                    ..
                } => {
                    break 'running;
                }
                Event::KeyDown {
                    keycode: Some(Keycode::Space),
                    ..
                } => {
                    utils::key_down(&mut key_manager, " ".to_string());
                }
                Event::KeyUp {
                    keycode: Some(Keycode::Space),
                    ..
                } => {
                    utils::key_up(&mut key_manager, " ".to_string());
                }
                Event::KeyDown { keycode, .. } => match keycode {
                    None => {}
                    Some(key) => {
                        utils::key_down(&mut key_manager, key.to_string());
                    }
                },
                Event::KeyUp { keycode, .. } => match keycode {
                    None => {}
                    Some(key) => {
                        utils::key_up(&mut key_manager, key.to_string());
                    }
                },
                _ => {}
            }
        }

        game::update(&mut gs.ecs, &mut key_manager);
        dispatcher.dispatch(&gs.ecs);
        gs.ecs.maintain();
        render(&mut canvas, &mut tex_man, &texture_creator, &font, &gs.ecs)?;

        // Time management
        ::std::thread::sleep(Duration::new(0, 1_000_000_000u32 / 60));
    }

    Ok(())
}
```

// ********** tutorial16 - components

// ********** tutorial16 - game

// ********** tutorial16 - missile

```
use specs::prelude::*;
use specs::{Entities, Join};

use crate::components;

pub struct MissileMover;

impl<'a> System<'a> for MissileMover {
    type SystemData = (
        WriteStorage<'a, components::Position>,
        WriteStorage<'a, components::Renderable>,
        ReadStorage<'a, components::Missile>,
        Entities<'a>,
    );

    fn run(&mut self, data: Self::SystemData) {
        let (mut positions, mut renderables, missiles, entities) = data;

        for (pos, rend, missile, entity) in
            (&mut positions, &mut renderables, &missiles, &entities).join()
        {
            let radian = pos.rot.to_radians();

            let move_x = missile.speed * radian.sin();
            let move_y = missile.speed * radian.cos();
            pos.x += move_x;
            pos.y -= move_y;
            if pos.x > crate::GAME_WIDTH.into()
                || pos.x < 0.0
                || pos.y > crate::GAME_HEIGHT.into()
                || pos.y < 0.0
            {
                entities.delete(entity).ok();
            }

            rend.rot = pos.rot;
        }
    }
}

pub struct MissileStriker;

impl<'a> System<'a> for MissileStriker {
    type SystemData = (
        WriteStorage<'a, components::Position>,
        WriteStorage<'a, components::Renderable>,
        WriteStorage<'a, components::Missile>,
        WriteStorage<'a, components::Asteroid>,
        WriteStorage<'a, components::Player>,
        WriteStorage<'a, components::GameData>,
        Entities<'a>,
    );

    fn run(&mut self, data: Self::SystemData) {
        let (positions, rends, missiles, asteroids, _players, _, entities) = &data;
        let mut asteroid_creation = Vec::<components::PendingAsteroid>::new();
        let mut score: u32 = 0;

        for (missile_pos, _, _, missile_entity) in (positions, rends, missiles, entities).join() {
            for (asteroid_pos, asteroid_rend, _, asteroid_entity) in
                (positions, rends, asteroids, entities).join()
            {
                let diff_x: f64 = (missile_pos.x - asteroid_pos.x).abs();
                let diff_y: f64 = (missile_pos.y - asteroid_pos.y).abs();
                let hyp: f64 = ((diff_x * diff_x) + (diff_y * diff_y)).sqrt();
                if hyp < asteroid_rend.o_w as f64 / 2.0 {
                    score += 10;
                    entities.delete(missile_entity).ok();
                    entities.delete(asteroid_entity).ok();
                    let new_size = asteroid_rend.o_w / 2;
                    if new_size >= 25 {
                        asteroid_creation.push(components::PendingAsteroid {
                            x: asteroid_pos.x,
                            y: asteroid_pos.y,
                            rot: asteroid_pos.rot - 90.0,
                            size: new_size,
                        });
                        asteroid_creation.push(components::PendingAsteroid {
                            x: asteroid_pos.x,
                            y: asteroid_pos.y,
                            rot: asteroid_pos.rot + 90.0,
                            size: new_size,
                        });
                    }
                }
            }
        }

        let (mut positions, mut rends, _, mut asteroids, _, _, entities) = data;
        for new_asteroid in asteroid_creation {
            let new_ast = entities.create();
            positions
                .insert(
                    new_ast,
                    components::Position {
                        x: new_asteroid.x,
                        y: new_asteroid.y,
                        rot: new_asteroid.rot,
                    },
                )
                .ok();
            asteroids
                .insert(
                    new_ast,
                    components::Asteroid {
                        speed: 2.5,
                        rot_speed: 0.5,
                    },
                )
                .ok();
            rends
                .insert(
                    new_ast,
                    components::Renderable {
                        tex_name: String::from("img/asteroid.png"),
                        i_w: 100,
                        i_h: 100,
                        o_w: new_asteroid.size,
                        o_h: new_asteroid.size,
                        frame: 0,
                        total_frames: 1,
                        rot: 0.0,
                    },
                )
                .ok();
        }

        let (_, _, _, _, _, mut gamedatas, _) = data;
        for gamedata in (&mut gamedatas).join() {
            gamedata.score += score;
            let mut gamestate = crate::GAMESTATE.lock().unwrap();
            if gamedata.score > gamestate.high_score {
                gamestate.high_score = gamedata.score;
            }
        }
    }
}
```

// ********** tutorial16 - texture_manager

// ********** tutorial16 - utils

```
// ******************************************************************************
// ********** tutorial17 - main
use sdl2::event::Event;
use sdl2::keyboard::Keycode;
use sdl2::pixels::Color;
use sdl2::render::{TextureCreator, WindowCanvas};
use sdl2::video::WindowContext;

use sdl2::rect::Point;
use sdl2::rect::Rect;

use specs::{DispatcherBuilder, Join, World, WorldExt};

use std::collections::HashMap;
use std::path::Path;
use std::sync::Mutex;
use std::time::Duration;

use once_cell::sync::Lazy;

pub mod asteroid;
pub mod components;
pub mod game;
pub mod missile;
pub mod texture_manager;
pub mod utils;

const GAME_WIDTH: u32 = 800;
const GAME_HEIGHT: u32 = 600;

fn render(
    canvas: &mut WindowCanvas,
    texture_manager: &mut texture_manager::TextureManager<WindowContext>,
    texture_creator: &TextureCreator<WindowContext>,
    font: &sdl2::ttf::Font,
    ecs: &World,
) -> Result<(), String> {
    //
    let color = Color::RGB(0, 0, 0);
    canvas.set_draw_color(color);
    canvas.clear();

    let positions = ecs.read_storage::<components::Position>();
    {
        canvas.set_draw_color(Color::RGBA(255, 255, 255, 128));

        let stars = ecs.read_storage::<components::Star>();
        for (pos, star) in (&positions, &stars).join() {
            canvas.fill_rect(Rect::new(pos.x as i32, pos.y as i32, star.size, star.size))?;
        }
    }

    let renderables = ecs.read_storage::<components::Renderable>();

    for (renderable, pos) in (&renderables, &positions).join() {
        let src = Rect::new(0, 0, renderable.i_w, renderable.i_h);
        let x: i32 = pos.x as i32;
        let y: i32 = pos.y as i32;
        let dest = Rect::new(
            x - ((renderable.o_w / 2) as i32),
            y - ((renderable.o_h / 2) as i32),
            renderable.o_w,
            renderable.o_h,
        );

        let center = Point::new((renderable.o_w / 2) as i32, (renderable.o_h / 2) as i32);
        let texture = texture_manager.load(&renderable.tex_name)?;
        canvas.copy_ex(
            &texture,
            src,            //source rect
            dest,           // dest rect
            renderable.rot, // angle
            center,         // center
            false,          // flip horizontal
            false,          // flip vertical
        )?;
    }

    let players = ecs.read_storage::<components::Player>();
    for (renderable, pos, _) in (&renderables, &positions, &players).join() {
        let src = Rect::new(0, 0, renderable.i_w, renderable.i_h);
        let x: i32 = pos.x as i32;
        let y: i32 = pos.y as i32;
        let mut dest = Rect::new(
            x - ((renderable.o_w / 2) as i32),
            y - ((renderable.o_h / 2) as i32),
            renderable.o_w,
            renderable.o_h,
        );

        let mut draw_second = false;
        if dest.x < (renderable.o_w / 2).try_into().unwrap() {
            dest.x += crate::GAME_WIDTH as i32;
            draw_second = true;
        } else if dest.x > (crate::GAME_WIDTH - renderable.o_w / 2) as i32 {
            dest.x -= crate::GAME_WIDTH as i32;
            draw_second = true;
        }
        if dest.y < (renderable.o_h / 2).try_into().unwrap() {
            dest.y += crate::GAME_HEIGHT as i32;
            draw_second = true;
        } else if dest.y > (crate::GAME_HEIGHT - renderable.o_h / 2) as i32 {
            dest.y -= crate::GAME_HEIGHT as i32;
            draw_second = true;
        }
        if !draw_second {
            break;
        }
        let center = Point::new((renderable.o_w / 2) as i32, (renderable.o_h / 2) as i32);
        let texture = texture_manager.load(&renderable.tex_name)?;
        canvas.copy_ex(
            &texture,
            src,            //source rect
            dest,           // dest rect
            renderable.rot, // angle
            center,         // center
            false,          // flip horizontal
            false,          // flip vertical
        )?;
    }

    let gamedatas = ecs.read_storage::<components::GameData>();
    for gamedata in (gamedatas).join() {
        // Show Score
        let score: String = "Score: ".to_string() + &gamedata.score.to_string();
        let surface = font
            .render(&score)
            .blended(Color::RGBA(255, 0, 0, 128))
            .map_err(|e| e.to_string())?;
        let texture = texture_creator
            .create_texture_from_surface(&surface)
            .map_err(|e| e.to_string())?;

        let target = Rect::new(10 as i32, 0 as i32, 100 as u32, 50 as u32);
        canvas.copy(&texture, None, Some(target))?;

        // Show Level
        let level: String = "Level: ".to_string() + &gamedata.level.to_string();
        let surface = font
            .render(&level)
            .blended(Color::RGBA(255, 0, 0, 128))
            .map_err(|e| e.to_string())?;
        let texture = texture_creator
            .create_texture_from_surface(&surface)
            .map_err(|e| e.to_string())?;

        let target = Rect::new(
            (crate::GAME_WIDTH - 110) as i32,
            0 as i32,
            100 as u32,
            50 as u32,
        );
        canvas.copy(&texture, None, Some(target))?;

        // High Score
        let high_score_value = &GAMESTATE.lock().unwrap().high_score;
        let high_score: String = "High Score: ".to_string() + &high_score_value.to_string();
        let surface = font
            .render(&high_score)
            .blended(Color::RGBA(255, 0, 0, 128))
            .map_err(|e| e.to_string())?;
        let texture = texture_creator
            .create_texture_from_surface(&surface)
            .map_err(|e| e.to_string())?;

        let target = Rect::new(
            10 as i32,
            (crate::GAME_HEIGHT - 60) as i32,
            100 as u32,
            50 as u32,
        );
        canvas.copy(&texture, None, Some(target))?;
    }

    canvas.present();
    Ok(())
}

struct State {
    ecs: World,
}

pub struct GameState {
    high_score: u32,
}

static GAMESTATE: Lazy<Mutex<GameState>> = Lazy::new(|| Mutex::new(GameState { high_score: 0 }));

fn main() -> Result<(), String> {
    println!("Starting Rusteroids");

    let sdl_context = sdl2::init()?;
    let video_subsystem = sdl_context.video()?;

    let window = video_subsystem
        .window("Rusteroids", GAME_WIDTH, GAME_HEIGHT)
        .position_centered()
        .build()
        .expect("could not initialize video subsystem");

    let mut canvas = window
        .into_canvas()
        .build()
        .expect("could not make a canvas");

    let texture_creator = canvas.texture_creator();
    let mut tex_man = texture_manager::TextureManager::new(&texture_creator);

    // Load the images before the main loop so we don't try and load during gameplay
    tex_man.load("img/space_ship.png")?;
    tex_man.load("img/asteroid.png")?;
    tex_man.load("img/missile.png")?;

    // Prepare fonts
    let ttf_context = sdl2::ttf::init().map_err(|e| e.to_string())?;
    let font_path: &Path = Path::new(&"fonts/OpenSans-Bold.ttf");
    let mut font = ttf_context.load_font(font_path, 128)?;
    font.set_style(sdl2::ttf::FontStyle::BOLD);

    let mut event_pump = sdl_context.event_pump()?;
    let mut key_manager: HashMap<String, bool> = HashMap::new();

    let mut gs = State { ecs: World::new() };
    gs.ecs.register::<components::Position>();
    gs.ecs.register::<components::Renderable>();
    gs.ecs.register::<components::Player>();
    gs.ecs.register::<components::Asteroid>();
    gs.ecs.register::<components::Missile>();
    gs.ecs.register::<components::GameData>();
    gs.ecs.register::<components::Star>();

    let mut dispatcher = DispatcherBuilder::new()
        .with(asteroid::AsteroidMover, "asteroid_mover", &[])
        .with(asteroid::AsteroidCollider, "asteroid_collider", &[])
        .with(missile::MissileMover, "missile_mover", &[])
        .with(missile::MissileStriker, "missile_striker", &[])
        .build();

    game::load_world(&mut gs.ecs);

    'running: loop {
        // Handle events
        for event in event_pump.poll_iter() {
            match event {
                Event::Quit { .. } => {
                    break 'running;
                }
                Event::KeyDown {
                    keycode: Some(Keycode::Escape),
                    ..
                } => {
                    break 'running;
                }
                Event::KeyDown {
                    keycode: Some(Keycode::Space),
                    ..
                } => {
                    utils::key_down(&mut key_manager, " ".to_string());
                }
                Event::KeyUp {
                    keycode: Some(Keycode::Space),
                    ..
                } => {
                    utils::key_up(&mut key_manager, " ".to_string());
                }
                Event::KeyDown { keycode, .. } => match keycode {
                    None => {}
                    Some(key) => {
                        utils::key_down(&mut key_manager, key.to_string());
                    }
                },
                Event::KeyUp { keycode, .. } => match keycode {
                    None => {}
                    Some(key) => {
                        utils::key_up(&mut key_manager, key.to_string());
                    }
                },
                _ => {}
            }
        }

        game::update(&mut gs.ecs, &mut key_manager);
        dispatcher.dispatch(&gs.ecs);
        gs.ecs.maintain();
        render(&mut canvas, &mut tex_man, &texture_creator, &font, &gs.ecs)?;

        // Time management
        ::std::thread::sleep(Duration::new(0, 1_000_000_000u32 / 60));
    }

    Ok(())
}
```

// ********** tutorial17 - asteroids

// ********** tutorial17 - components

// ********** tutorial17 - game

// ********** tutorial17 - missile

// ********** tutorial17 - texture_manager

// ********** tutorial17 - utils

```
// ********************************************************************************
// ********** tutorial18 - main
use sdl2::event::Event;
use sdl2::keyboard::Keycode;
use sdl2::pixels::Color;
use sdl2::render::{TextureCreator, WindowCanvas};
use sdl2::video::WindowContext;

use sdl2::rect::Point;
use sdl2::rect::Rect;

use specs::{DispatcherBuilder, Join, World, WorldExt};

use std::collections::HashMap;
use std::path::Path;
use std::sync::Mutex;
use std::time::Duration;

use once_cell::sync::Lazy;

pub mod asteroid;
pub mod components;
pub mod game;
pub mod missile;
pub mod sound_manager;
pub mod texture_manager;
pub mod utils;

const GAME_WIDTH: u32 = 800;
const GAME_HEIGHT: u32 = 600;

fn render(
    canvas: &mut WindowCanvas,
    texture_manager: &mut texture_manager::TextureManager<WindowContext>,
    texture_creator: &TextureCreator<WindowContext>,
    font: &sdl2::ttf::Font,
    ecs: &World,
) -> Result<(), String> {
    //
    let color = Color::RGB(0, 0, 0);
    canvas.set_draw_color(color);
    canvas.clear();

    let positions = ecs.read_storage::<components::Position>();
    {
        canvas.set_draw_color(Color::RGBA(255, 255, 255, 128));

        let stars = ecs.read_storage::<components::Star>();
        for (pos, star) in (&positions, &stars).join() {
            canvas.fill_rect(Rect::new(pos.x as i32, pos.y as i32, star.size, star.size))?;
        }
    }

    let renderables = ecs.read_storage::<components::Renderable>();

    for (renderable, pos) in (&renderables, &positions).join() {
        let src = Rect::new(0, 0, renderable.i_w, renderable.i_h);
        let x: i32 = pos.x as i32;
        let y: i32 = pos.y as i32;
        let dest = Rect::new(
            x - ((renderable.o_w / 2) as i32),
            y - ((renderable.o_h / 2) as i32),
            renderable.o_w,
            renderable.o_h,
        );

        let center = Point::new((renderable.o_w / 2) as i32, (renderable.o_h / 2) as i32);
        let texture = texture_manager.load(&renderable.tex_name)?;
        canvas.copy_ex(
            &texture,
            src,            //source rect
            dest,           // dest rect
            renderable.rot, // angle
            center,         // center
            false,          // flip horizontal
            false,          // flip vertical
        )?;
    }

    let players = ecs.read_storage::<components::Player>();
    for (renderable, pos, _) in (&renderables, &positions, &players).join() {
        let src = Rect::new(0, 0, renderable.i_w, renderable.i_h);
        let x: i32 = pos.x as i32;
        let y: i32 = pos.y as i32;
        let mut dest = Rect::new(
            x - ((renderable.o_w / 2) as i32),
            y - ((renderable.o_h / 2) as i32),
            renderable.o_w,
            renderable.o_h,
        );

        let mut draw_second = false;
        if dest.x < (renderable.o_w / 2).try_into().unwrap() {
            dest.x += crate::GAME_WIDTH as i32;
            draw_second = true;
        } else if dest.x > (crate::GAME_WIDTH - renderable.o_w / 2) as i32 {
            dest.x -= crate::GAME_WIDTH as i32;
            draw_second = true;
        }
        if dest.y < (renderable.o_h / 2).try_into().unwrap() {
            dest.y += crate::GAME_HEIGHT as i32;
            draw_second = true;
        } else if dest.y > (crate::GAME_HEIGHT - renderable.o_h / 2) as i32 {
            dest.y -= crate::GAME_HEIGHT as i32;
            draw_second = true;
        }
        if !draw_second {
            break;
        }

        let center = Point::new((renderable.o_w / 2) as i32, (renderable.o_h / 2) as i32);
        let texture = texture_manager.load(&renderable.tex_name)?;
        canvas.copy_ex(
            &texture,
            src,            //source rect
            dest,           // dest rect
            renderable.rot, // angle
            center,         // center
            false,          // flip horizontal
            false,          // flip vertical
        )?;
    }

    let gamedatas = ecs.read_storage::<components::GameData>();
    for gamedata in (gamedatas).join() {
        // Show Score
        let score: String = "Score: ".to_string() + &gamedata.score.to_string();
        let surface = font
            .render(&score)
            .blended(Color::RGBA(255, 0, 0, 128))
            .map_err(|e| e.to_string())?;
        let texture = texture_creator
            .create_texture_from_surface(&surface)
            .map_err(|e| e.to_string())?;

        let target = Rect::new(10 as i32, 0 as i32, 100 as u32, 50 as u32);
        canvas.copy(&texture, None, Some(target))?;

        // Show Level
        let level: String = "Level: ".to_string() + &gamedata.level.to_string();
        let surface = font
            .render(&level)
            .blended(Color::RGBA(255, 0, 0, 128))
            .map_err(|e| e.to_string())?;
        let texture = texture_creator
            .create_texture_from_surface(&surface)
            .map_err(|e| e.to_string())?;

        let target = Rect::new(
            (crate::GAME_WIDTH - 110) as i32,
            0 as i32,
            100 as u32,
            50 as u32,
        );
        canvas.copy(&texture, None, Some(target))?;

        // High Score
        let high_score_value = &GAMESTATE.lock().unwrap().high_score;
        let high_score: String = "High Score: ".to_string() + &high_score_value.to_string();
        let surface = font
            .render(&high_score)
            .blended(Color::RGBA(255, 0, 0, 128))
            .map_err(|e| e.to_string())?;
        let texture = texture_creator
            .create_texture_from_surface(&surface)
            .map_err(|e| e.to_string())?;

        let target = Rect::new(
            10 as i32,
            (crate::GAME_HEIGHT - 60) as i32,
            100 as u32,
            50 as u32,
        );
        canvas.copy(&texture, None, Some(target))?;
    }

    canvas.present();
    Ok(())
}

struct State {
    ecs: World,
}

pub struct GameState {
    high_score: u32,
}

static GAMESTATE: Lazy<Mutex<GameState>> = Lazy::new(|| Mutex::new(GameState { high_score: 0 }));

fn main() -> Result<(), String> {
    println!("Starting Rusteroids");

    let sdl_context = sdl2::init()?;
    let video_subsystem = sdl_context.video()?;

    let window = video_subsystem
        .window("Rusteroids", GAME_WIDTH, GAME_HEIGHT)
        .position_centered()
        .build()
        .expect("could not initialize video subsystem");

    let mut canvas = window
        .into_canvas()
        .build()
        .expect("could not make a canvas");

    let texture_creator = canvas.texture_creator();
    let mut tex_man = texture_manager::TextureManager::new(&texture_creator);

    // Load the images before the main loop so we don't try and load during gameplay
    tex_man.load("img/space_ship.png")?;
    tex_man.load("img/asteroid.png")?;
    tex_man.load("img/missile.png")?;

    let mut sound_manager = sound_manager::SoundManager::new();
    // Load Sounds to prevent loading during gameplay
    sound_manager.load_sound(&"sounds/fx/missile.ogg".to_string());

    // Prepare fonts
    let ttf_context = sdl2::ttf::init().map_err(|e| e.to_string())?;
    let font_path: &Path = Path::new(&"fonts/OpenSans-Bold.ttf");
    let mut font = ttf_context.load_font(font_path, 128)?;
    font.set_style(sdl2::ttf::FontStyle::BOLD);

    let mut event_pump = sdl_context.event_pump()?;
    let mut key_manager: HashMap<String, bool> = HashMap::new();

    let mut gs = State { ecs: World::new() };
    gs.ecs.register::<components::Position>();
    gs.ecs.register::<components::Renderable>();
    gs.ecs.register::<components::Player>();
    gs.ecs.register::<components::Asteroid>();
    gs.ecs.register::<components::Missile>();
    gs.ecs.register::<components::GameData>();
    gs.ecs.register::<components::Star>();
    gs.ecs.register::<components::SoundCue>();

    let mut dispatcher = DispatcherBuilder::new()
        .with(asteroid::AsteroidMover, "asteroid_mover", &[])
        .with(asteroid::AsteroidCollider, "asteroid_collider", &[])
        .with(missile::MissileMover, "missile_mover", &[])
        .with(missile::MissileStriker, "missile_striker", &[])
        .build();

    game::load_world(&mut gs.ecs);

    'running: loop {
        // Handle events
        for event in event_pump.poll_iter() {
            match event {
                Event::Quit { .. } => {
                    break 'running;
                }
                Event::KeyDown {
                    keycode: Some(Keycode::Escape),
                    ..
                } => {
                    break 'running;
                }
                Event::KeyDown {
                    keycode: Some(Keycode::Space),
                    ..
                } => {
                    utils::key_down(&mut key_manager, " ".to_string());
                }
                Event::KeyUp {
                    keycode: Some(Keycode::Space),
                    ..
                } => {
                    utils::key_up(&mut key_manager, " ".to_string());
                }
                Event::KeyDown { keycode, .. } => match keycode {
                    None => {}
                    Some(key) => {
                        utils::key_down(&mut key_manager, key.to_string());
                    }
                },
                Event::KeyUp { keycode, .. } => match keycode {
                    None => {}
                    Some(key) => {
                        utils::key_up(&mut key_manager, key.to_string());
                    }
                },
                _ => {}
            }
        }

        game::update(&mut gs.ecs, &mut key_manager);
        dispatcher.dispatch(&gs.ecs);
        gs.ecs.maintain();

        let cues = gs.ecs.write_storage::<components::SoundCue>();
        let entities = gs.ecs.entities();

        for (cue, entity) in (&cues, &entities).join() {
            sound_manager.play_sound(cue.filename.to_string());
            entities.delete(entity).ok();
        }

        render(&mut canvas, &mut tex_man, &texture_creator, &font, &gs.ecs)?;

        // Time management
        ::std::thread::sleep(Duration::new(0, 1_000_000_000u32 / 60));
    }

    Ok(())
}
```

// ********** tutorial18 - asteroids

// ********** tutorial18 - components

```
use specs::prelude::*;
use specs_derive::Component;
use vector2d::Vector2D;

#[derive(Component)]
pub struct Position {
    pub x: f64,
    pub y: f64,
    pub rot: f64,
}

#[derive(Component)]
pub struct Renderable {
    pub tex_name: String,
    pub i_w: u32,
    pub i_h: u32,
    pub o_w: u32,
    pub o_h: u32,
    pub frame: u32,
    pub total_frames: u32,
    pub rot: f64,
}

#[derive(Component)]
pub struct Player {
    pub impulse: Vector2D<f64>,   // The next impulse to add to the speed
    pub cur_speed: Vector2D<f64>, // The current speed of the player
}

#[derive(Component)]
pub struct Asteroid {
    pub speed: f64,
    pub rot_speed: f64,
}
```

```
#[derive(Component)]
pub struct Missile {
    pub speed: f64,
}

pub struct PendingAsteroid {
    pub x: f64,
    pub y: f64,
    pub rot: f64,
    pub size: u32,
}

#[derive(Component)]
pub struct GameData {
    pub score: u32,
    pub level: u32,
}

#[derive(Component)]
pub struct Star {
    pub size: u32,
}
```

```
#[derive(Component)]
pub struct SoundCue {
    pub filename: String,
}
```

// ******** tutorial18 - game**

```
use rand::Rng;
use specs::{Builder, Join, World, WorldExt};
use std::collections::HashMap;
use vector2d::Vector2D;

use crate::components;
use crate::utils;

const ROTATION_SPEED: f64 = 1.5;
const PLAYER_SPEED: f64 = 4.5;

pub fn update(ecs: &mut World, key_manager: &mut HashMap<String, bool>) {
    // Check status of game world
    let mut must_reload_world = false;
    let mut current_player_position = components::Position {
        x: 0.0,
        y: 0.0,
        rot: 0.0,
    };
    {
        let players = ecs.read_storage::<components::Player>();
        let positions = ecs.read_storage::<components::Position>();

        for (pos, _player) in (&positions, &players).join() {
            current_player_position.x = pos.x;
            current_player_position.y = pos.y;
        }

        if players.join().count() < 1 {
            must_reload_world = true;
        }
    }

    if must_reload_world {
        // Remove all of the previous entities so we can start again
        ecs.delete_all();
        // Reset the world to first state
        load_world(ecs);
    }

    // Check if all asteroids are missing
    let mut must_create_asteroid = false;
    let mut number_asteroids: u32 = 0;
    {
        let asteroids = ecs.read_storage::<components::Asteroid>();
        if asteroids.join().count() < 1 {
            must_create_asteroid = true;

            let mut gamedatas = ecs.write_storage::<components::GameData>();
            for gamedata in (&mut gamedatas).join() {
                gamedata.level += 1;
                number_asteroids = (gamedata.level / 3) + 1;
            }
        }
    }

    if must_create_asteroid {
        let mut asteroid_count: u32 = 0;
        while asteroid_count < number_asteroids {
            let mut rng = rand::thread_rng();
            let next_x = rng.gen_range(50.0..(crate::GAME_WIDTH as f64 - 50.0));
            let next_y = rng.gen_range(50.0..(crate::GAME_HEIGHT as f64 - 50.0));
            let next_rot = rng.gen_range(0.0..360.0);

            let diff_x = (current_player_position.x - next_x).abs();
            let diff_y = (current_player_position.y - next_y).abs();
            if ((diff_x * diff_x) + (diff_y * diff_y)).sqrt() < 150.0 {
                // We are too close to the player
                continue;
            }
            asteroid_count += 1;
            let new_asteroid = components::Position {
                x: next_x,
                y: next_y,
                rot: next_rot,
            };
            create_asteroid(ecs, new_asteroid, 100);
        }
    }

    let mut player_pos = components::Position {
        x: 0.0,
        y: 0.0,
        rot: 0.0,
    };
    let mut must_fire_missile = false;
    {
        let mut positions = ecs.write_storage::<components::Position>();
        let mut players = ecs.write_storage::<components::Player>();
        let mut renderables = ecs.write_storage::<components::Renderable>();

        for (player, pos, renderable) in (&mut players, &mut positions, &mut renderables).join() {
            if crate::utils::is_key_pressed(&key_manager, "D") {
                pos.rot += ROTATION_SPEED;
            }
            if crate::utils::is_key_pressed(&key_manager, "A") {
                pos.rot -= ROTATION_SPEED;
            }
            update_movement(pos, player);
            if crate::utils::is_key_pressed(&key_manager, "W") {
                let radians = pos.rot.to_radians();

                let move_x = PLAYER_SPEED * radians.sin();
                let move_y = PLAYER_SPEED * radians.cos();
                let move_vec = Vector2D::<f64>::new(move_x, move_y);

                player.impulse += move_vec;
            }

            if pos.rot > 360.0 {
                pos.rot -= 360.0;
            }
            if pos.rot < 0.0 {
                pos.rot += 360.0;
            }

            if pos.x > crate::GAME_WIDTH.into() {
                pos.x -= crate::GAME_WIDTH as f64;
            }
            if pos.x < 0.0 {
                pos.x += crate::GAME_WIDTH as f64;
            }
            if pos.y > crate::GAME_HEIGHT.into() {
                pos.y -= crate::GAME_HEIGHT as f64;
            }
            if pos.y < 0.0 {
                pos.y += crate::GAME_HEIGHT as f64;
            }

            if utils::is_key_pressed(&key_manager, " ") {
                utils::key_up(key_manager, " ".to_string());
                must_fire_missile = true;
                player_pos.x = pos.x;
                player_pos.y = pos.y;
                player_pos.rot = pos.rot;
            }

            // Update the graphic to reflect the rotation
            renderable.rot = pos.rot;
        }
    }

    if must_fire_missile {
        fire_missile(ecs, player_pos);
    }
}

const MAX_SPEED: f64 = 3.5;
const FRICTION: f64 = 0.99;

pub fn update_movement(pos: &mut components::Position, player: &mut components::Player) {
    player.cur_speed *= FRICTION;

    player.cur_speed += player.impulse;
    if player.cur_speed.length() > MAX_SPEED {
        player.cur_speed = player.cur_speed.normalise();
        player.cur_speed = player.cur_speed * MAX_SPEED;
    }

    pos.x += player.cur_speed.x;
    pos.y -= player.cur_speed.y;

    player.impulse = vector2d::Vector2D::new(0.0, 0.0);
}

pub const NUMBER_OF_STARS: u32 = 45;

pub fn load_world(ecs: &mut World) {
    ecs.create_entity()
        .with(components::Position {
            x: 350.0,
            y: 250.0,
            rot: 0.0,
        })
        .with(components::Renderable {
            tex_name: String::from("img/space_ship.png"),
            i_w: 100,
            i_h: 100,
            o_w: 50,
            o_h: 50,
            frame: 0,
            total_frames: 1,
            rot: 0.0,
        })
        .with(components::Player {
            impulse: vector2d::Vector2D::new(0.0, 0.0),
            cur_speed: vector2d::Vector2D::new(0.0, 0.0),
        })
        .build();

    create_asteroid(
        ecs,
        components::Position {
            x: 400.0,
            y: 235.0,
            rot: 45.0,
        },
        50,
    );

    ecs.create_entity()
        .with(components::GameData { score: 0, level: 1 })
        .build();

    for _ in 0..=NUMBER_OF_STARS {
        let mut rng = rand::thread_rng();
        let star_x = rng.gen_range(0.0..crate::GAME_WIDTH as f64);
        let star_y = rng.gen_range(0.0..crate::GAME_HEIGHT as f64);
        let size = rng.gen_range(1..=3);

        ecs.create_entity()
            .with(components::Position {
                x: star_x,
                y: star_y,
                rot: 0.0,
            })
            .with(components::Star { size: size })
            .build();
    }
}

pub fn create_asteroid(ecs: &mut World, position: components::Position, asteroid_size: u32) {
    ecs.create_entity()
        .with(position)
        .with(components::Renderable {
            tex_name: String::from("img/asteroid.png"),
            i_w: 100,
            i_h: 100,
            o_w: asteroid_size,
            o_h: asteroid_size,
            frame: 0,
            total_frames: 1,
            rot: 0.0,
        })
        .with(crate::components::Asteroid {
            speed: 2.5,
            rot_speed: 0.5,
        })
        .build();
}

const MAX_MISSILES: usize = 5;

fn fire_missile(ecs: &mut World, position: components::Position) {
    {
        let missiles = ecs.read_storage::<components::Missile>();
        if missiles.count() > MAX_MISSILES - 1 {
            return;
        }
    }
    ecs.create_entity()
        .with(position)
        .with(components::Renderable {
            tex_name: String::from("img/missile.png"),
            i_w: 50,
            i_h: 100,
            o_w: 10,
            o_h: 20,
            frame: 0,
            total_frames: 1,
            rot: 0.0,
        })
        .with(components::Missile { speed: 5.0 })
        .build();

    ecs.create_entity()
        .with(components::SoundCue {
            filename: "sounds/fx/missile.ogg".to_string(),
        })
        .build();
}
```

// ******** tutorial18 - missile**

// ******** tutorial18 - sound_manager**

```
use kira::{
    manager::{backend::cpal::CpalBackend, AudioManager, AudioManagerSettings},
    sound::static_sound::StaticSoundData,
};
use std::collections::HashMap;

pub struct SoundManager {
    pub sound_manager: AudioManager<CpalBackend>,
    pub sounds: HashMap<String, StaticSoundData>,
}

impl SoundManager {
    pub fn new() -> Self {
        Self {
            sound_manager: AudioManager::<CpalBackend>::new(AudioManagerSettings::default())
                .expect("Failed to load Kira Audio Engine"),
            sounds: HashMap::new(),
        }
    }

    pub fn play_sound(&mut self, filename: String) {
        if self.sounds.contains_key(&filename) {
            if let Some(x) = self.sounds.get_mut(&filename) {
                self.sound_manager
                    .play(x.clone())
                    .expect("Failed to play sounds");
            }
        } else {
            println!("Sound doesn't exist");
        }
    }

    pub fn load_sound(&mut self, filename: &String) {
        self.sounds
            .entry((&filename).to_string())
            .or_insert(StaticSoundData::from_file(filename).expect("Failed to load sound"));
    }
}
```

```
// ********** tutorial18 - texture_manager
```

```
// ********** tutorial18 - utils
```