Detección y resolución de colisiones usando Rust

Agustin Escobar - https://github.com/agustinesco

Contexto

- Champions of mirra

Contexto

- Champions of mirra
- Estructura del backend principalmente en Elixir

```elixir
defmodule Physics do
  @moduledoc """
  Physics
  """
  use Rustler,
    otp_app: :arena,
    crate: :physics

  def check_collisions(_entity, _entities), do: :erlang.nif_error(:nif_not_loaded)
```

Contexto

- Champions of mirra
- Estructura del backend principalmente en Elixir
- ¿Por qué incluir rust?
  - Rustler
  - Módulo de físicas
  - Estructura de entidades

```elixir
defmodule Physics do
  @moduledoc """
  Physics
  """
  use Rustler,
    otp_app: :arena,
    crate: :physics

  def check_collisions(_entity, _entities), do: :erlang.nif_error(:nif_not_loaded)
```

Contexto

Contexto

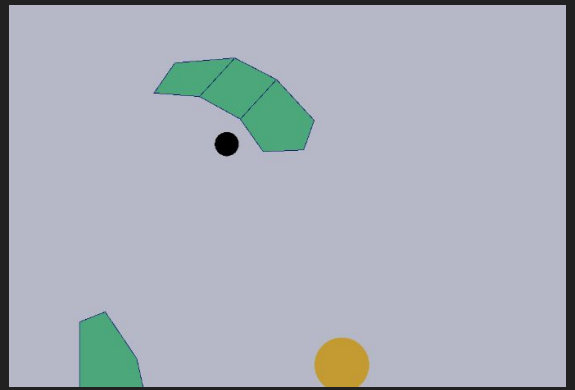
- Detección de colisiones
  - Jeffrey Thompson

Contexto

- Detección de colisiones
  - Jeffrey Thompson
  - Tiempo algorítmico: O(Z * N)
    - Z: Cantidad de polígonos
    - N: Cantidad de vertices del poligono

Contexto

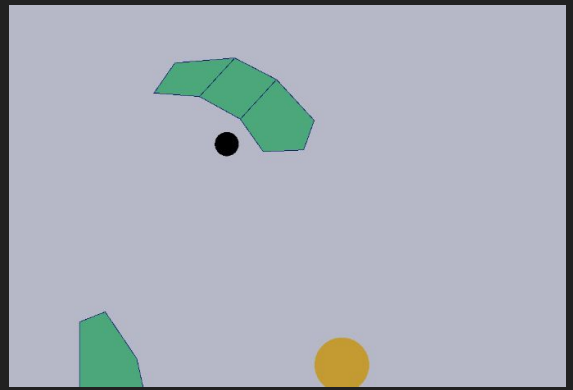
- Detección de colisiones
  - Jeffrey Thompson
  - Tiempo algorítmico: O(Z * N)
    - Z: Cantidad de polígonos
    - N: Cantidad de vertices del poligono
- Resolución de colisiones
  - S.A.T.



Representación 2d de un obstáculo y un jugador

Contexto

- Detección de colisiones
  - Jeffrey Thompson
  - Tiempo algorítmico: O(Z * N)
    - Z: Cantidad de polígonos
    - N: Cantidad de vertices del poligono
- Resolución de colisiones
  - S.A.T.
  - Tiempo algorítmico: O(Z * N^2)
    - Z: Cantidad de polígonos
    - N: Cantidad de vertices del poligono



Representación 2d de un obstáculo y un jugador

Contexto

- Movimiento de los jugadores

```rust
fn move_entity_to_closest_available_position(
    entity: &mut Entity,
    external_wall: &Entity,
    obstacles: &HashMap<u64, Entity>,
) {
    if entity.category == Category::Player && !entity.is_inside_map(external_wall) {
        entity.move_to_next_valid_position_inside(external_wall);
    }

    let collides_with: Vec<u64> = entity.collides_with(entities: &obstacles.clone().into_values().collect());

    if entity.category == Category::Player && !collides_with.is_empty() {
        let collided_with: Vec<&Entity> = collides_with Vec<u64>
            .iter() Iter<u64>
            .map(|id: &u64| obstacles.get(id).unwrap()) impl Iterator<Item = &Entity>
            .collect();
        entity.move_to_next_valid_position_outside(collided_with, obstacles, external_wall);
    }
}
```

Contexto

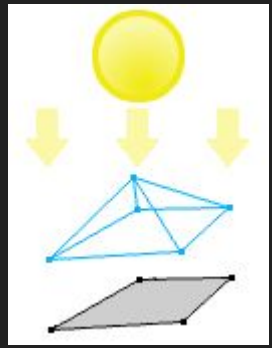
- Movimiento de los jugadores
- Cuando entra S.A.T.

```rust
pub fn move_to_next_valid_position_outside(
    &mut self,
    collided_with: Vec<&Entity>,
    obstacles: &HashMap<u64, Entity>,
    external_wall: &Entity,
) {
    for entity: &Entity in collided_with {
        match entity.shape {
            Shape::Circle => {…
            Shape::Polygon => {
                let (collided: bool, direction: Position, depth: f32) =
                    intersect_circle_polygon(circle: self, polygon: entity, obstacles, external_wall);

                if collided {
                    let new_pos: Position = Position {
                        x: self.position.x + direction.x * depth,
                        y: self.position.y + direction.y * depth,
                    };
                    self.position = new_pos;
                }
            }
            _ => continue,
        }
    }
} fn move_to_next_valid_position_outside
```

S.A.T. Separated Axis Theorem

Si dos objetos convexos no están solapados, eso significa que existe un axis en el cual la proyección de esos objetos no están solapando.

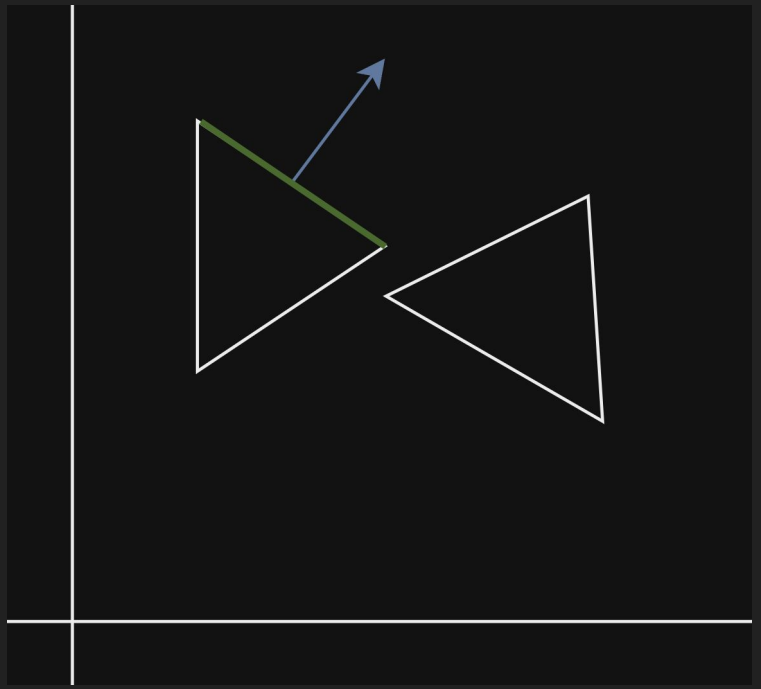
- Que es la proyección



S.A.T. Separated Axis Theorem

Si dos objetos convexos no están solapados, eso significa que existe un axis en el cual la proyección de esos objetos no están solapando.

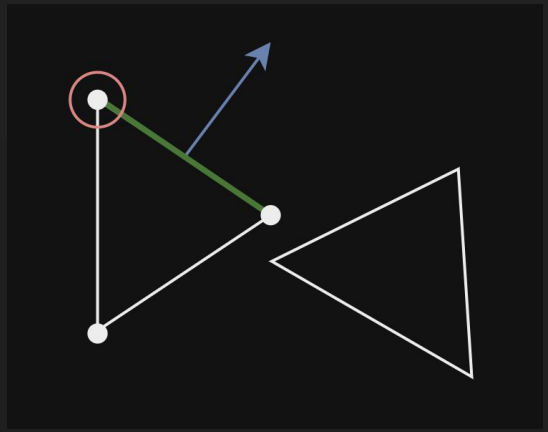
- Que es la proyección
- Definir axis a proyectar
    - Elegir una normal



S.A.T. Separated Axis Theorem

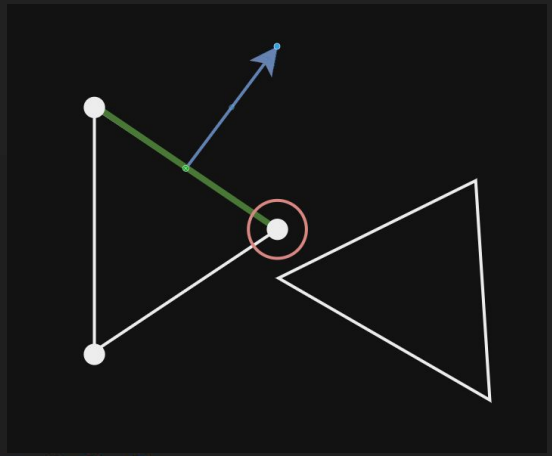
```
// Check normal and depth for polygon
for current_vertex_index: usize in 0..polygon.vertices.len() {
    let mut next_vertex_index: usize = current_vertex_index + 1;
    if next_vertex_index == polygon.vertices.len() {
        next_vertex_index = 0
    };
    let current_vertex: Position = polygon.vertices[current_vertex_index];
    let next_vertex: Position = polygon.vertices[next_vertex_index];

    let current_line: Position = Position::sub(a: &current_vertex, b: &next_vertex);
    // the axis will be the perpendicular line drawn from the current line of the polygon
    axis = Position {
        x: -current_line.y,
        y: current_line.x,
    };

    axis.normalize();
```

S.A.T. Separated Axis Theorem



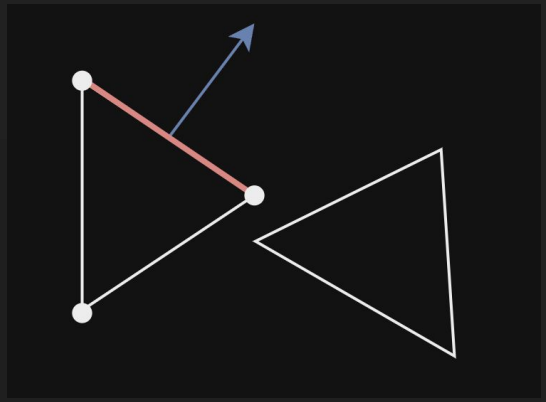
```
// Check normal and depth for polygon
for current_vertex_index: usize in 0..polygon.vertices.len() {
    let mut next_vertex_index: usize = current_vertex_index + 1;
    if next_vertex_index == polygon.vertices.len() {
        next_vertex_index = 0
    };
    let current_vertex: Position = polygon.vertices[current_vertex_index];
    let next_vertex: Position = polygon.vertices[next_vertex_index];

    let current_line: Position = Position::sub(a: &current_vertex, b: &next_vertex);
    // the axis will be the perpendicular line drawn from the current line of the polygon
    axis = Position {
        x: -current_line.y,
        y: current_line.x,
    };

    axis.normalize();
```

S.A.T. Separated Axis Theorem



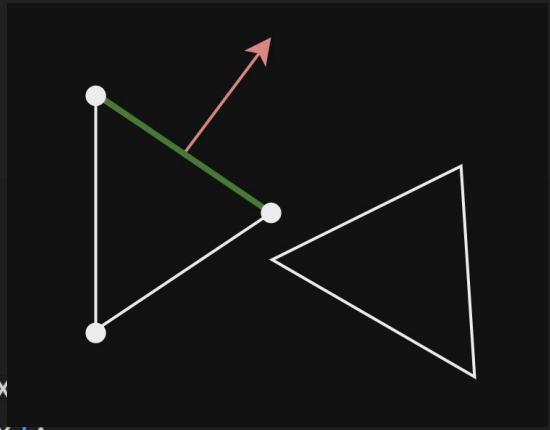
```
// Check normal and depth for polygon
for current_vertex_index: usize in 0..polygon.vertices.len() {
    let mut next_vertex_index: usize = current_vertex_index + 1;
    if next_vertex_index == polygon.vertices.len() {
        next_vertex_index = 0
    };
    let current_vertex: Position = polygon.vertices[current_vertex_index];
    let next_vertex: Position = polygon.vertices[next_vertex_index];

    let current_line: Position = Position::sub(a: &current_vertex, b: &next_vertex);
    // the axis will be the perpendicular line drawn from the current line of the polygon
    axis = Position {
        x: -current_line.y,
        y: current_line.x,
    };

    axis.normalize();
```

S.A.T. Separated Axis Theorem



```
// Check normal and depth for polygon
for current_vertex_index: usize in 0..polygon.vertices.len() {
    let mut next_vertex_index: usize = current_vertex_index + 1;
    if next_vertex_index == polygon.vertices.len() {
        next_vertex_index = 0
    };
    let current_vertex: Position = polygon.vertices[current_vertex_index];
    let next_vertex: Position = polygon.vertices[next_vertex_index];

    let current_line: Position = Position::sub(a: &current_vertex, b: &next_vertex);
    // the axis will be the perpendicular line drawn from the current line of the polygon
    axis = Position {
        x: -current_line.y,
        y: current_line.x,
    };

    axis.normalize();
```

S.A.T. Separated Axis Theorem

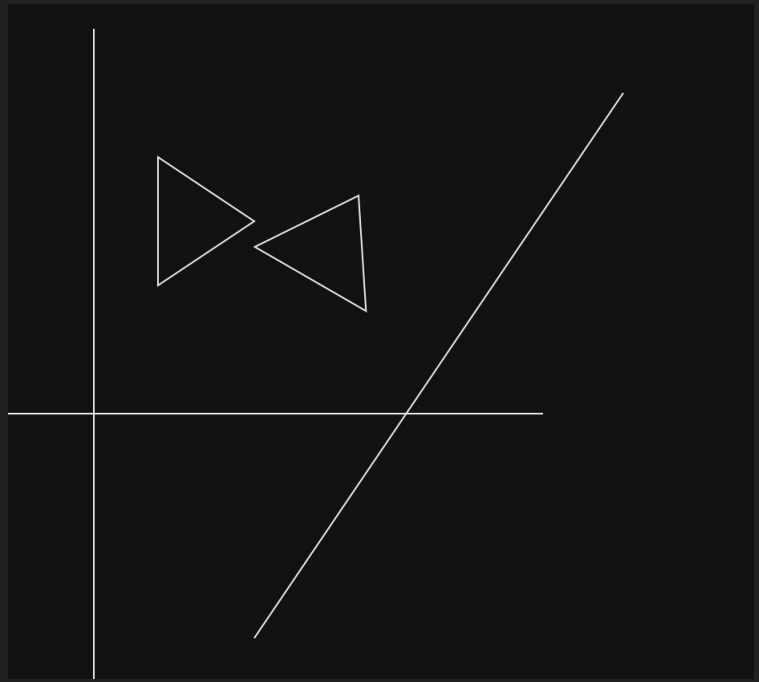


```
let mut next_vertex_index: usize = current_vertex_index + 1;
if next_vertex_index == polygon.vertices.len() {
    next_vertex_index = 0
};
let current_vertex: Position = polygon.vertices[current_vertex
let next_vertex: Position = polygon.vertices[next_vertex_index];

let current_line: Position = Position::sub(a: &current_vertex, b: &next_vertex);
// the axis will be the perpendicular line drawn from the current line of the polygon
axis = Position {
    x: -current_line.y,
    y: current_line.x,
};

axis.normalize();
```

S.A.T. Separated Axis Theorem

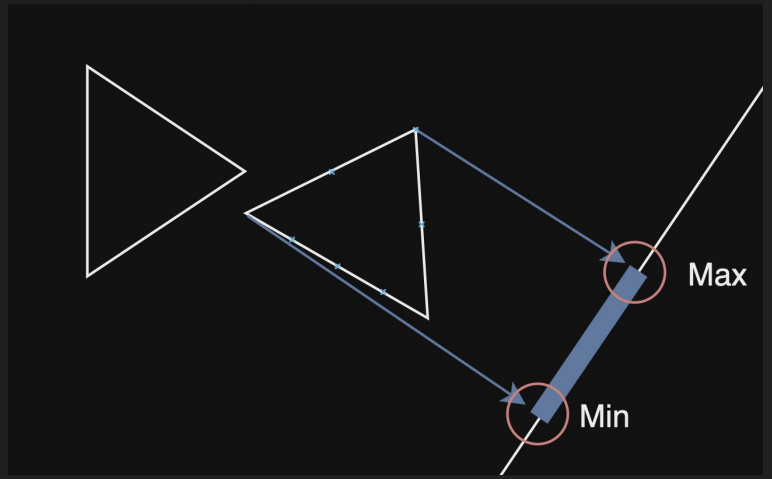
Si dos objetos convexos no están solapados, eso significa que existe un axis en el cual la proyección de esos objetos no están solapando.

- Que es la proyección
- Definir axis a proyectar
    - Elegir una normal



S.A.T. Separated Axis Theorem

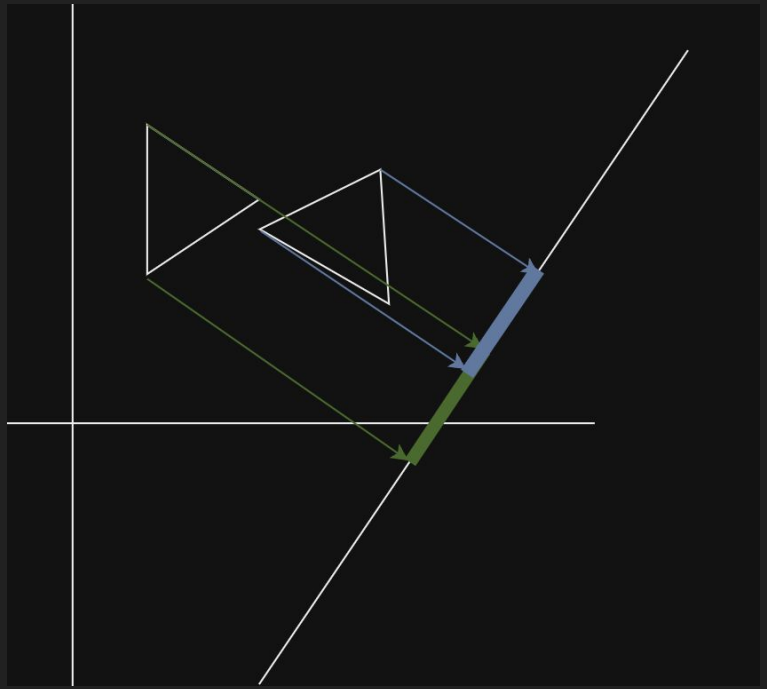
```rust
// Get the min and max values from a polygon projected on a specific axis
fn project_vertices(vertices: &Vec<Position>, axis: Position) -> (f32, f32) {
    let mut min: f32 = f32::MAX;
    let mut max: f32 = f32::MIN;

    for current: &Position in vertices {
        let projection: f32 = dot(a: current, b: axis);

        if projection < min {
            min = projection
        };
        if projection > max {
            max = projection
        };
    }

    (min, max)
}
```



S.A.T. Separated Axis Theorem

Si dos objetos convexos no están solapados, eso significa que existe un axis en el cual la proyección de esos objetos no están solapando.

- Que es la proyección
- Definir axis a proyectar
  - Elegir una normal
  - Proyección de sombras usando producto escalar

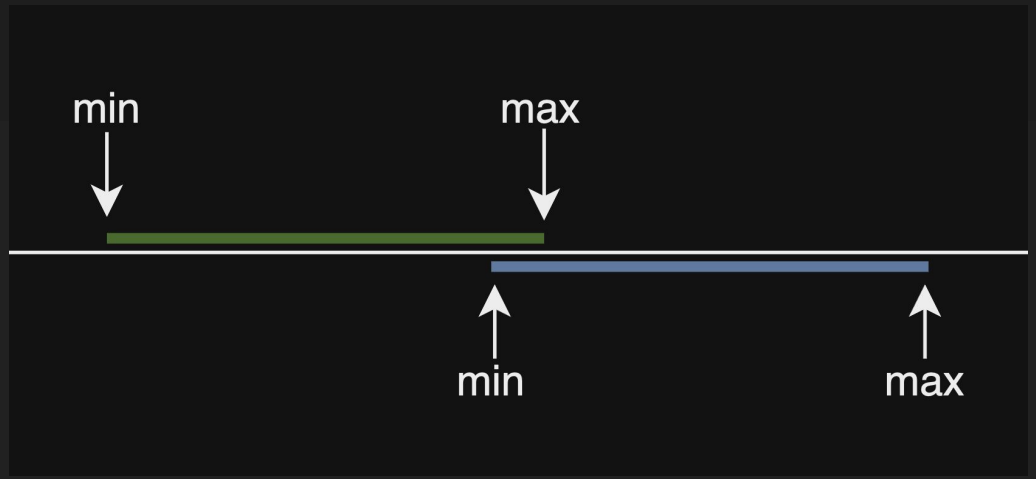


S.A.T. Separated Axis Theorem

```
let (min_polygon_cast_point: f32, max_polygon_cast_point: f32) = project_vertices(&polygon.vertices, axis);
let (min_circle_cast_point: f32, max_circle_cast_point: f32) = project_circle(circle, axis);

// If there's a gap between the polygon it means they do not collide and we can safely return false
if min_polygon_cast_point ⩾ max_circle_cast_point
    || min_circle_cast_point ⩾ max_polygon_cast_point
{
    return (false, normal, result_depth);
}
```

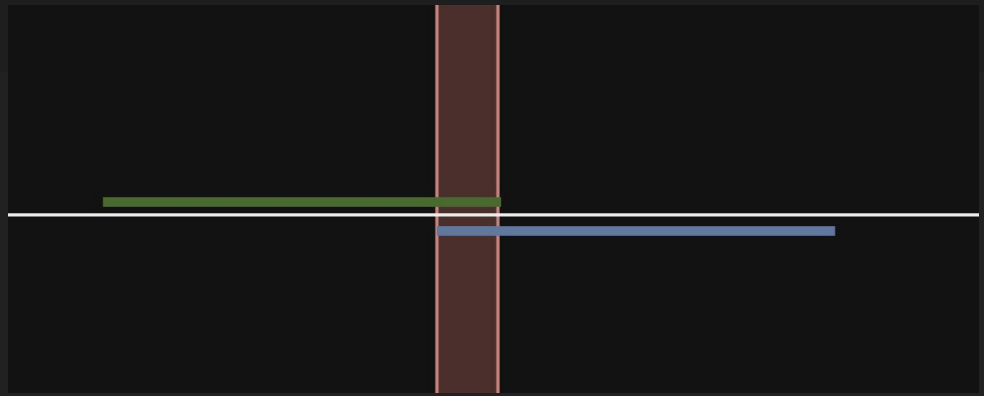
S.A.T. Separated Axis Theorem

```rust
let (min_polygon_cast_point: f32, max_polygon_cast_point: f32) = project_vertices(&polygon.vertices, axis);
let (min_circle_cast_point: f32, max_circle_cast_point: f32) = project_circle(circle, axis);

// If there's a gap between the polygon it means they do not collide and we can safely return false
if min_polygon_cast_point ⩾ max_circle_cast_point
    || min_circle_cast_point ⩾ max_polygon_cast_point
{
    return (false, normal, result_depth);
}
```



S.A.T. Separated Axis Theorem

```
let (min_polygon_cast_point: f32, max_polygon_cast_point: f32) = project_vertices(&polygon.vertices, axis);
let (min_circle_cast_point: f32, max_circle_cast_point: f32) = project_circle(circle, axis);

// If there's a gap between the polygon it means they do not collide and we can safely return false
if min_polygon_cast_point ⩾ max_circle_cast_point
    || min_circle_cast_point ⩾ max_polygon_cast_point
{
    return (false, normal, result_depth);
}
```



S.A.T. Separated Axis Theorem

```
let (min_polygon_cast_point: f32, max_polygon_cast_point: f32) = project_vertices(&polygon.vertices, axis);
let (min_circle_cast_point: f32, max_circle_cast_point: f32) = project_circle(circle, axis);

// If there's a gap between the polygon it means they do not collide and we can safely return false
if min_polygon_cast_point ⩾ max_circle_cast_point
    || min_circle_cast_point ⩾ max_polygon_cast_point
{
    return (false, normal, result_depth);
}
```

S.A.T. Separated Axis Theorem

Acumular la normal y la profundidad con el menor solapamiento

```
let circle_overlap_depth: f32 = max_circle_cast_point - min_polygon_cast_point;

let polygon_overlap_depth: f32 = max_polygon_cast_point - min_circle_cast_point;

let min_depth: f32 = f32::min(self: circle_overlap_depth, other: polygon_overlap_depth);

if min_depth < result_depth {
    normal = axis;
    result_depth = min_depth;
}
```

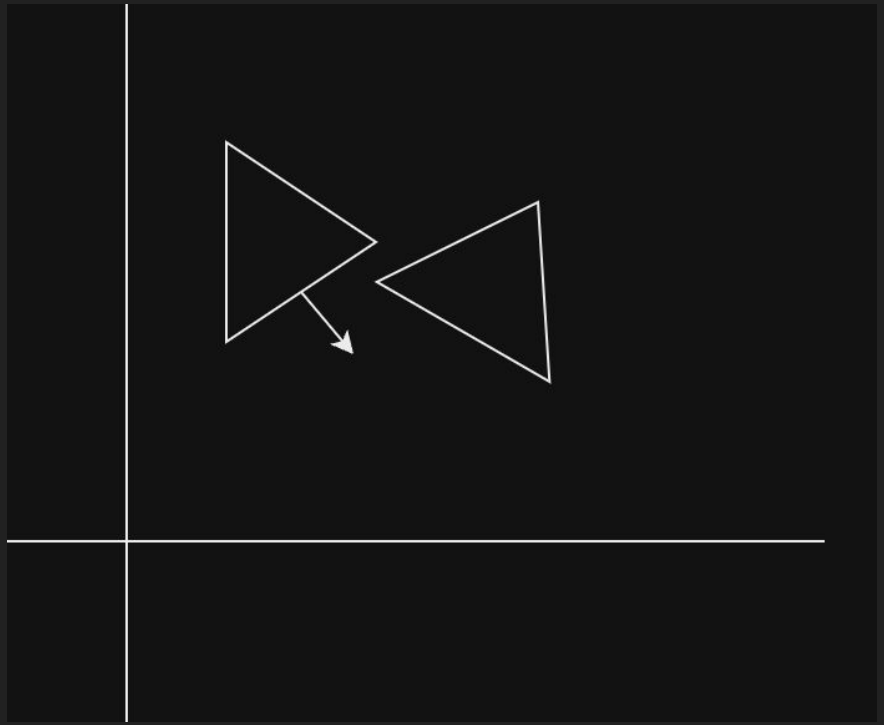
S.A.T. Separated Axis Theorem

Al finalizar de iterar todos los pares de vértices de ambos polígonos o el círculo se devuelve la normal con el menor solapamiento

```
    return (true, normal, result_depth);
} fn intersect_circle_polygon
```

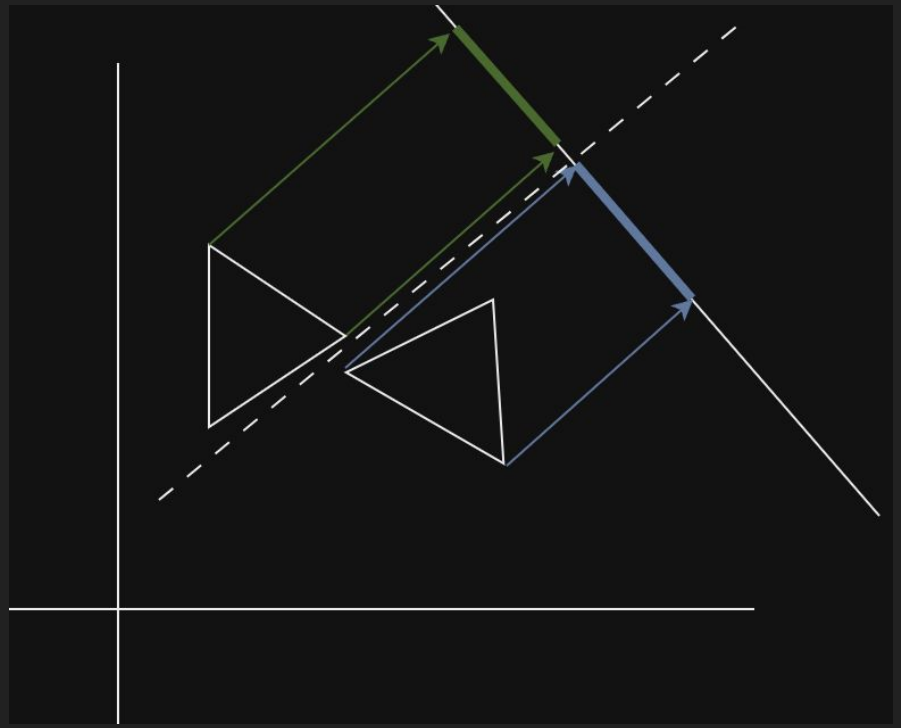
S.A.T. Separated Axis Theorem

Ejemplo de axis separador:



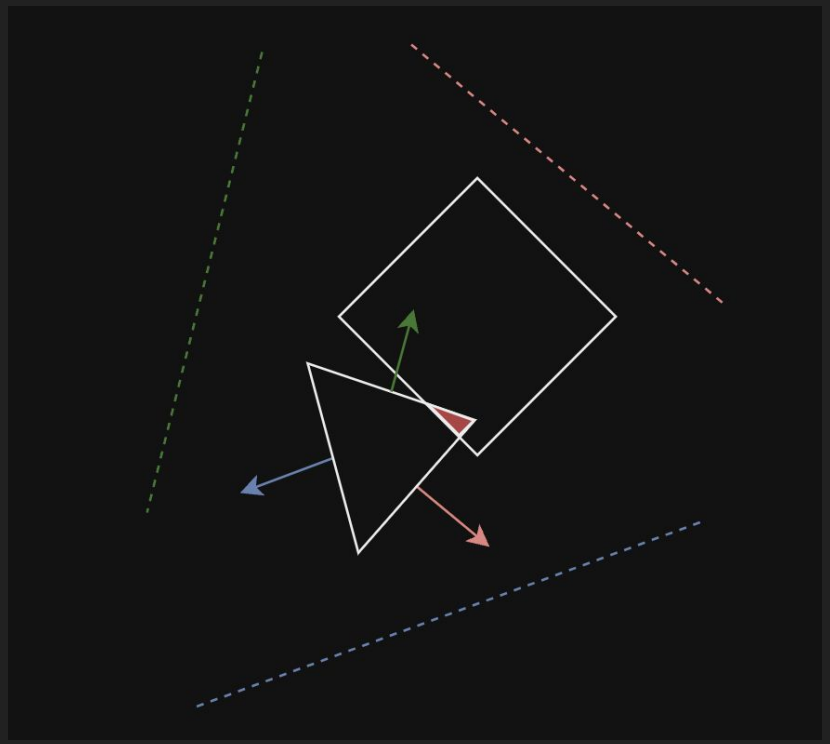
S.A.T. Separated Axis Theorem

Ejemplo de axis separador:



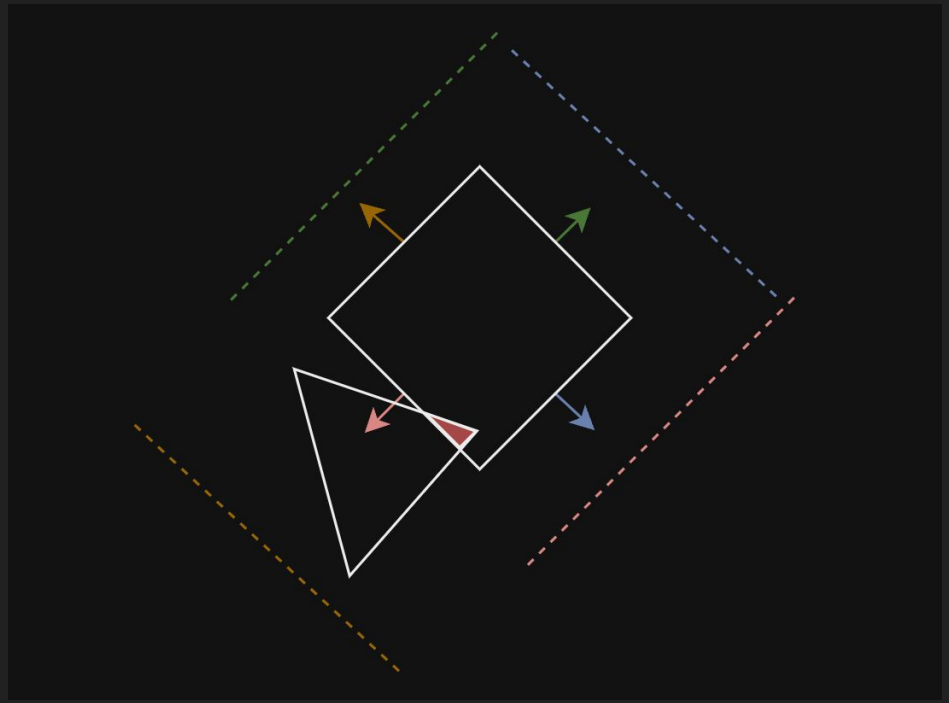
S.A.T. Separated Axis Theorem

Ejemplo de colisión:



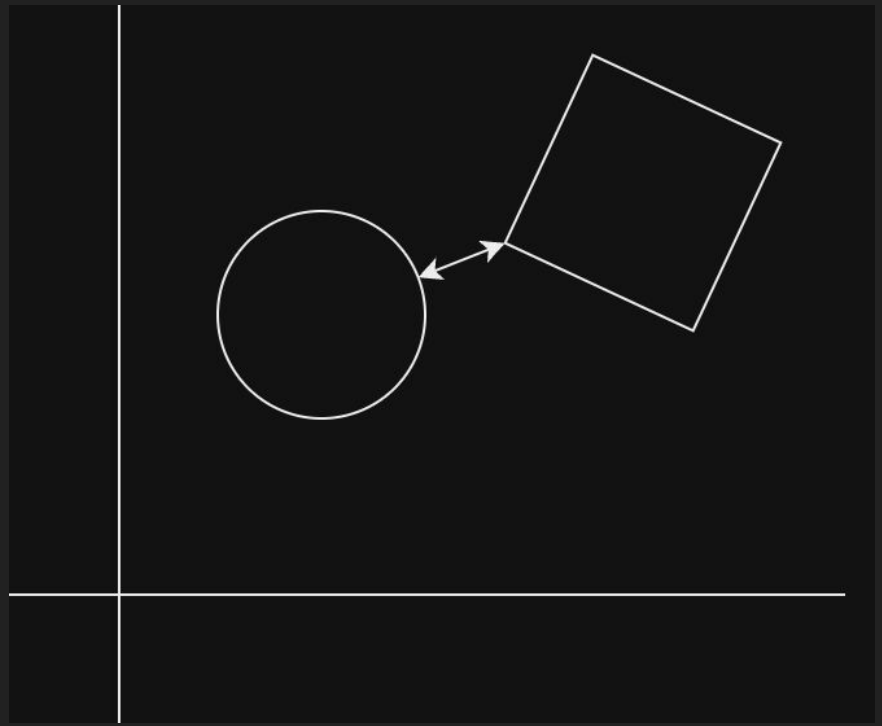
S.A.T. Separated Axis Theorem

Ejemplo de colisión:



S.A.T. Separated Axis Theorem

Ejemplo de axis con círculo:

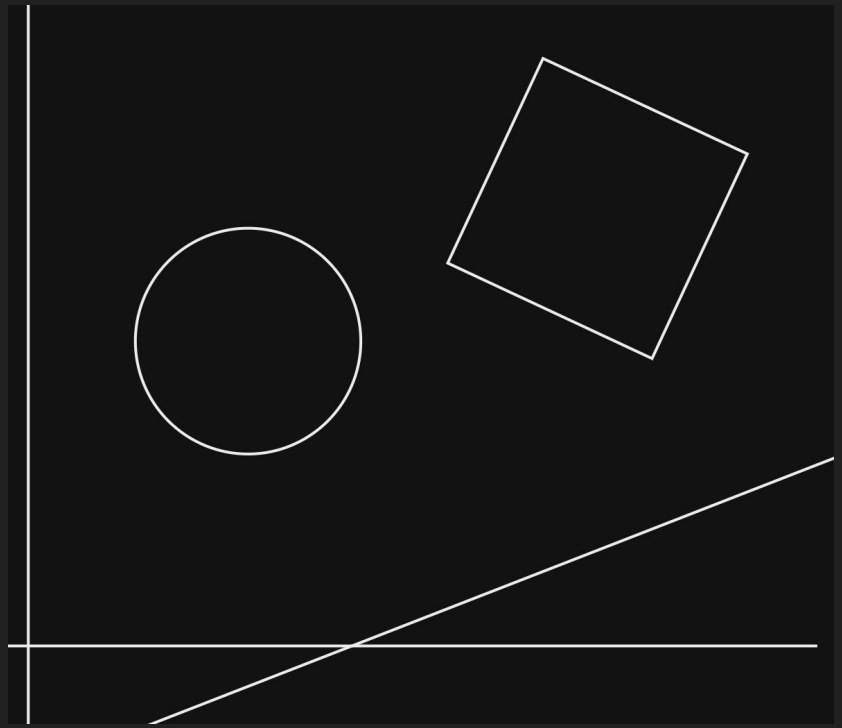


S.A.T. Separated Axis Theorem

```
// Check normal and depth for center
let closest_vertex: Position = find_closest_vertex(center: &circle.position, &polygon.vertices);
axis = Position::sub(a: &closest_vertex, b: &circle.position);
axis.normalize();
```

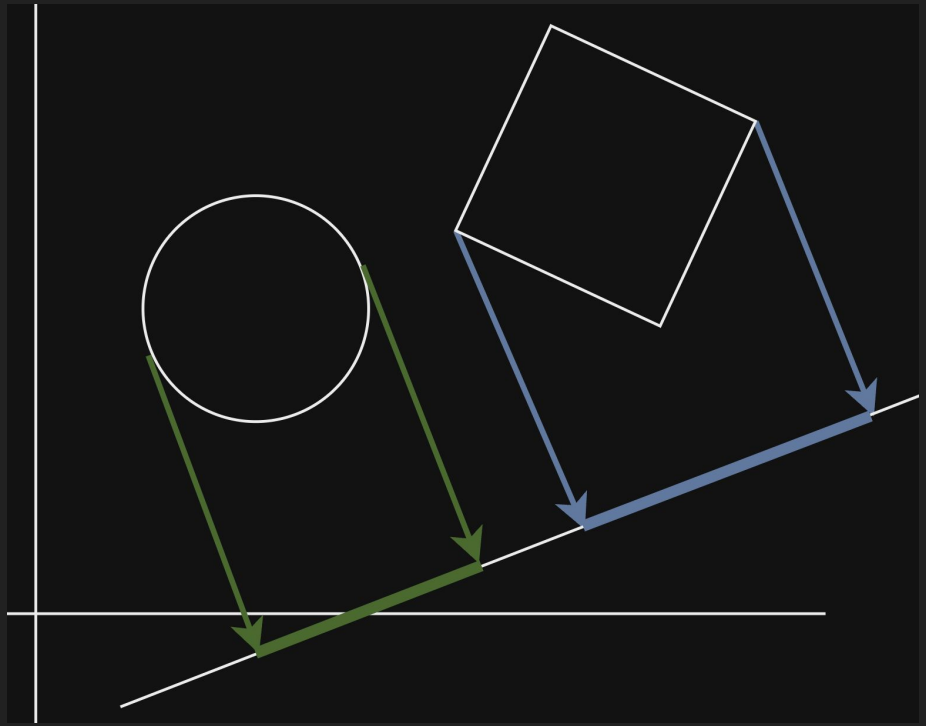
S.A.T. Separated Axis Theorem

Ejemplo de axis con círculo:



S.A.T. Separated Axis Theorem

Ejemplo de axis con círculo:



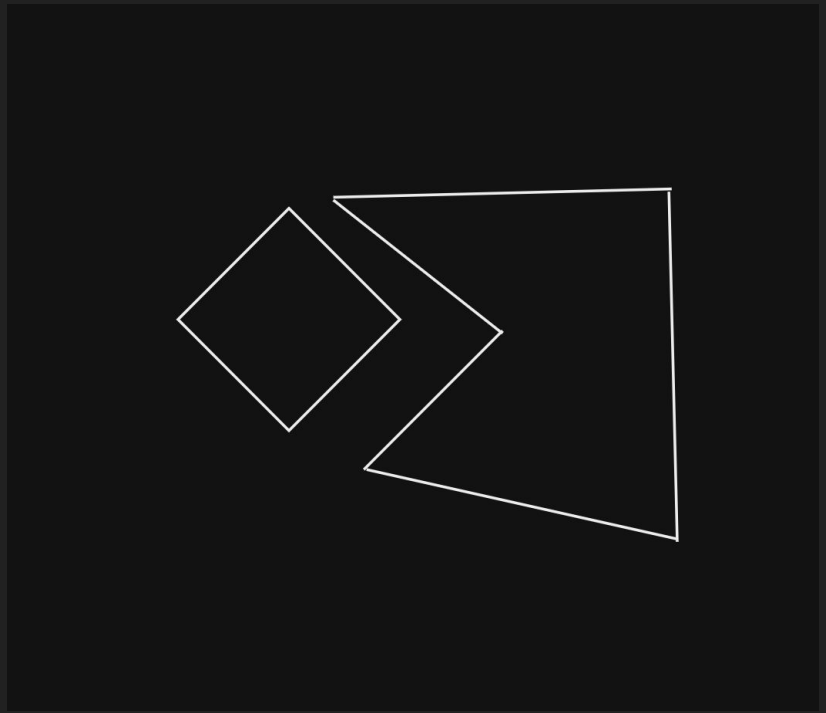
S.A.T. Separated Axis Theorem

Ejemplo de axis con círculo:

```
let (min_polygon_cast_point: f32, max_polygon_cast_point: f32) = project_vertices(&polygon.vertices, axis);
let (min_circle_cast_point: f32, max_circle_cast_point: f32) = project_circle(circle, axis);

// If there's a gap between the polygon it means they do not collide and we can safely return false
if min_polygon_cast_point ⩾ max_circle_cast_point
    || min_circle_cast_point ⩾ max_polygon_cast_point
{
    return (false, normal, result_depth);
}
```

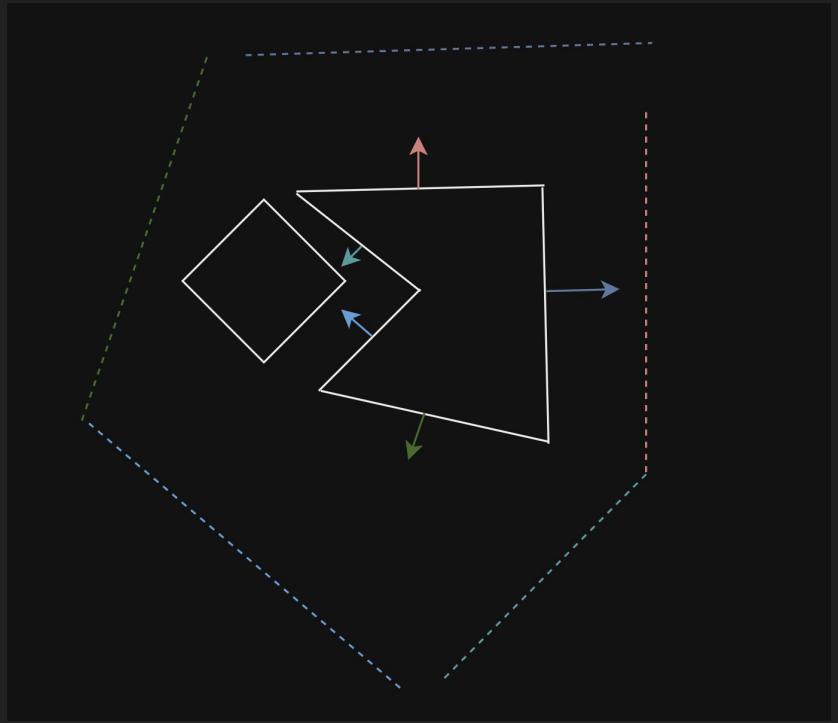
S.A.T. Separated Axis Theorem

Ejemplo de problema concavidad:



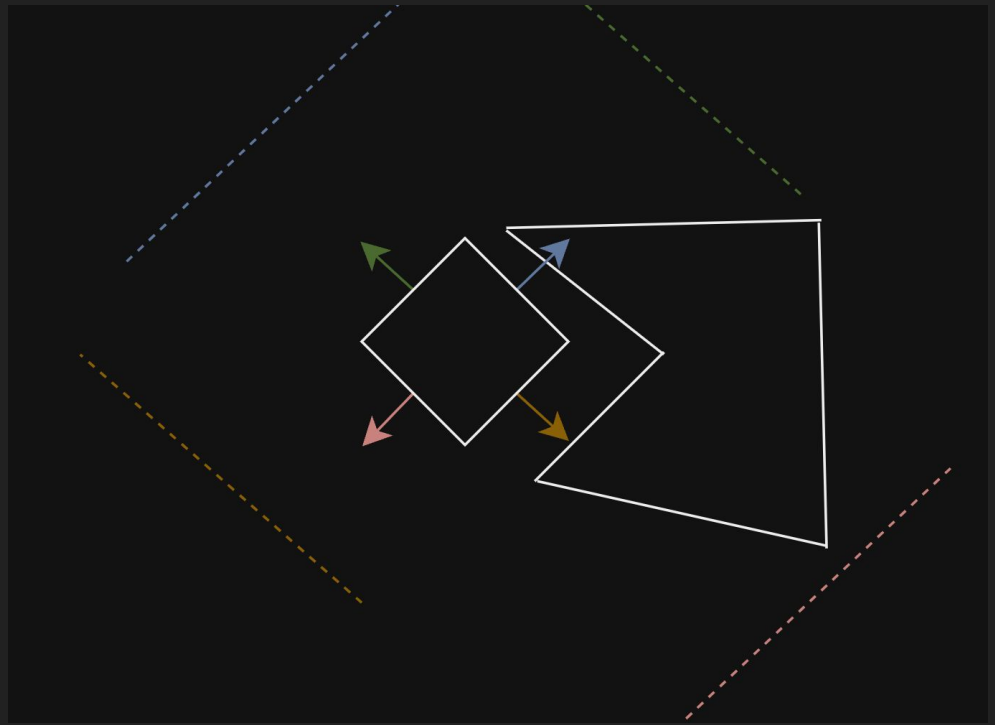
S.A.T. Separated Axis Theorem

Ejemplo de problema concavidad:



S.A.T. Separated Axis Theorem

Ejemplo de problema concavidad:



Conclusiones

- Dificultad de implementación
  - Líneas de elixir 3993
  - Líneas de rust 738 en total
    - Algoritmo de jeffrey 193 líneas
    - SAT 224 líneas

- Benchmark de distintos algoritmos
  - jeffrey 20 ms en promedio
  - SAT 9 ms en promedio

Recursos

- Video de youtube sat implementation https://www.youtube.com/watch?v=Zgf1DYrmSnk
- SAT article https://dyn4j.org/2010/01/sat/