CS550 Framework Technical Design Document

Josu Cubero

Spring 2020

Abstract

The goal of this subject is to build a 3D physics library aimed to be used in real time applications. This document, covers the details behind the theory and implementation of various physics topics, including rigid body dynamics, collision detection and collision resolution.

1 Rigid Body

The rigid body is the main container used to represent entities in the physics engine. As such, it contains all the relevant data that is going to be used as input and output for the pipeline.

In this project, the rigid body is based on a model[1] that presents unconstrained linear and angular movement induced by external forces. This forces affect the linear and angular momentum of the bodies modifying their velocities, position and orientation.

```
struct RigidBody
{
    float mass

    mat3 I_body
    mat3 I_inv_body

    vec3 position
    quat rotation
    vec3 linear_velocity
    vec3 angular_velocity
    vec3 linear_momentum
    vec3 angular_momentum

    add_force(const vec3 force_pos, const vec3 force_dir)
    integrate(const float dt)
}
```

Having all that data it is possible to simulate rigid bodies moving freely in a 3D space. As for the integration, the external forces will modify the momentum of the body and then they will be integrated to modify the velocities as well as the position and rotation.

```
add_force(const vec3 force_pos, const vec3 force_dir)
{
    linear_momentum += force_dir / mass
    vec3 torque = cross(force_pos - position, force_dir)
    angular_momentum += torque
}

integrate(const float dt)
{
    linear_velocity = linear_momentum / mass
    position += linear_velocity * dt
    mat3 I_inv = rotation * I_inv_body * transpose(rotation)
    angular_velocity = I_inv * angular_momentum
    quat d_rot = 1/2 * quat(0, angular_velocity) * rotation
    rotation = normalize(rotation + d_rot * dt)
}
```

2 Half Edge Mesh

To represent the shape of an object we use a modified version of a conventional physical mesh. This mesh, has additional functionalities that are useful for some of the algorithms further in the pipeline.

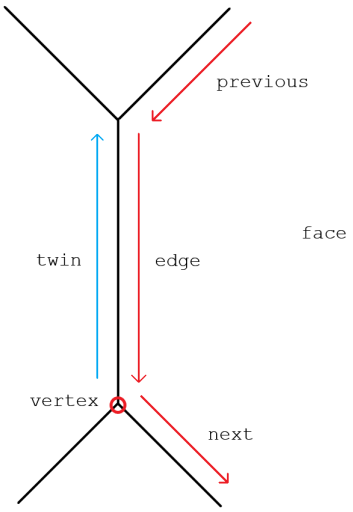

Figure 1: Half Edge representation

This mesh, stores the vertices and the faces of the shape. The faces save the data of one of its edges that is connected to the previous and next edges form

the same face in counter clockwise order, as well as its twin facing the opposite direction from the neighbor face. The half edges also store information about the vertex they are pointing to an the face they are contained in.

```
struct HalfEdge
{
    int vertex_index

    HalfEdge* next
    HalfEdge* prev
    HalfEdge* twin

    HalfEdgeFace* face
}
```

Constructing the half edge mesh from a set of vertices and three indices per face is fairly simple. However, with many shapes we may encounter coplanar faces. While it does not affect correctness, we may want to merge this faces to save computation cost in future steps.

Merging coplanar faces can be done by looping through every face of the shape and checking if any of the neighbour faces has the same normal vector. to do so, we use the half edges of the face and check the faces of the twin edge. If the normal vectors match, we will remove both, the edge and the twin and reconnect the previous and next edges accordingly to merge the faces.

3 Collision Detection

The algorithm used to detect if two convex polyhedron are intersecting is the separating axis theorem (SAT)[3]. This algorithm, as the name suggests, tries to find a separating axis between the convex polyhedron. If no axis is found the bodies must be colliding. The algorithm relies on skipping computations when a separating axis between the bodies is found. Therefore, the throwback of SAT is the computation cost of generating the contact information. For that reason, when used for real time applications it works specially well when there are a lot of entities but very few of them are colliding.

Given two convex polyhedron A and B, to determine if the bodies are colliding the algorithm searches for a separating axis in three steps.

The first two steps use the normal vectors of the faces of one of the bodies and check the most extreme vertex of the other body in the opposite direction. If the penetration is negative for any face, there is a separating axis. In order to compute the most extreme point, we will make use of the Hill Climbing Algorithm.

The third step checks if the cross product of any combination of edges in the shapes is a separating axis. Computing it using the brute force approach, would be too expensive. Instead, we are going to make use of Gauss maps to check if the pair of edges generates a Minkowski Difference (potential axis of separation) and reduce the amount of computations drastically.

3.1 Hill Climbing Algorithm

Making used of the Half Edge Mesh, we can compute the most extreme vertex in a given direction much faster than using a brute force approach.

This algorithm works in model space by taking a random vertex in the mesh as an starting point, then, comparing the distance from the origin to the mentioned point and all the neighbours and lastly, taking the point with the maximum distance as the starting point for the next iteration.

The reason why the half edge mesh comes in handy for this algorithm is because it allows us to easily iterate through the neighbor vertices by making use of the edges connecting them.

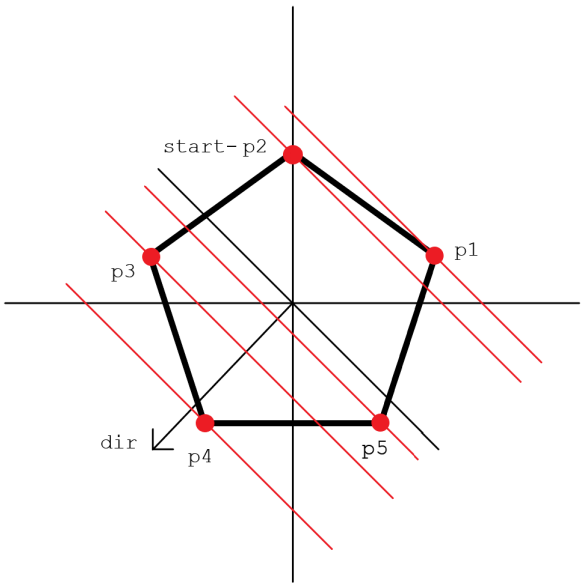

Figure 2: Hill Climbing Example

In the figure above we can see how starting the algorithm from the vertex $p2$, we then compare the distances from the origin for $p1$ and $p3$, $p3$ being the largest one. Repeating the process taking $p3$ as the starting vertex will lead us to $p4$ as the most extreme point in the direction of dir. Lastly, comparing the distances between $p3$, $p4$ and $p5$ we can conclude that $p4$ is the most extreme point.

The Hill Climbing Algorithm proofs to improve the performance of the program when utilized in complex shapes with many vertices.

3.2 Gauss Map

Gauss Maps are used to map a mesh into a unit sphere. The normal vector of the faces are represented as points in the unit spheres and the edges of the mesh are shown as arcs connecting those points.

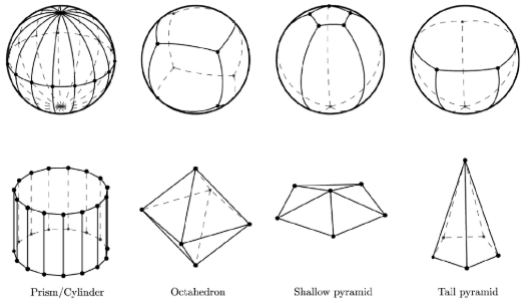

Figure 3: Gauss Map Examples [2]

3.3 Minkowski Difference

The Minkowski Difference checks if to convex polyhedron are intersecting by subtracting all the vertices from one shape with all the vertices from the other shape and creating the convex hull of all the points. If the origin of the world is contained in the Minkowski difference, the bodies are intersecting.

3.4 Edge Detection

Having the Gauss maps of both bodies mapped to the same unit sphere, it is possible to check which edge arcs overlap and therefore, generate a face of the Minkowski difference.

To check if the arcs associated to the edges in the Gauss map are intersection we build planes through both arcs and check if they intersect performing a triple product. Not only, that, but it also must be considered that the arcs belong to the same hemisphere on the Gauss map.

```
bool generate_minkowski_face(HalfEdge edge_A, HalfEdge edge_B)
{
    vec3 a =  edge_A.face.normal, b =  edge_A.twin.face.normal
    vec3 c = -edge_B.face.normal, d = -edge_B.twin.face.normal

    vec3 b_cross_a = cross(b, a)
    vec3 d_cross_c = cross(d, c)

    float cba = bxa * c, dba = bxa * d
    float adc = dxc * a, bdc = dxc * b

    return cba * dba < 0 && adc * bdc < 0 && cba * bdc > 0
}
```

Notice that the cross products between the face normal vectors can be skipped by computing the direction vectors of the edges.

3.5 SAT Solution

With all that information, SAT can be reduced to the following code:

```
bool overlap(const RigidBody A, const RigidBody B)
{
    if (has_separating_axis_face(A, B))
        return false

    if (has_separating_axis_face(B, A))
        return false

    if (has_separating_axis_edge(A, B))
        return false

    return true
}


bool has_separating_axis_face(const RigidBody A, const RigidBody B)
{
    for (face : A.faces)
    {
        vec3 support = B.hill_climbing(-face.normal)
        vec3 dir = support - face.point
        float separation = dir * face.normal

        if (separation > 0)
            return true
    }
    return false
}


bool has_separating_axis_edge(const RigidBody A, const RigidBody B)
{
    for (edge_A : A.edges)
    {
        for (edge_B : B.edges)
        {
            if (generate_minkowski_face(edge_A, edge_B))
            {
                float separation = distance(edge_A, edge_B)
                if(separation > 0)
                    return true
            }
        }
    }
    return false
}
```

3.6 Contact Generation

Once the Separating Axis Theorem determines that the two bodies are colliding, we want to get the contact information generated from that collision. To generate the contact manifold the penetration closest to zero from the previous checks is used.

When the smallest penetration occurs in the edge-edge case, the only information needed is the distance between the edges and the point from one edge closest to the the other edge.

The case for solving face-attribute however, is more challenging. The solution requires to find the incident face. This is done by looking for the most antinormal reference face of the opposite body. Then, the incident face must be clipped with the neighbor faces of the reference face.

After clipping the incident face, the points outside the reference face must be removed. The penetration would be determined by the point with the highest distance.

3.7 Sutherland Hodgeman

This algorithm is used to clip a polygon with another polygon. The idea is to clip each segment of the polygon with the clipping faces using segment-plane intersections. Then add the points to the output based on the following criteria A being the first point and B the second point in the segment traversing the polygon in counter-clockwise order.

A	B	out
Front	Front	B
Behind	Front	I, B
Front	Behind	I
Behind	Behind	X

4 Collision Resolution

The goal of resolving a collision is to separate bodies that are intersecting. The two approaches covered in this document make use of impulses, immediate forces, applied in the contact points of the collision.

4.1 Naive Solution

The naive approach applies an impulse in the direction of the normal vector of the contact. This scalar is computed by multiplying the effective mass with the relative velocity of the contact point respect to each body and the restitution factor.

$$m_e = \frac{1}{\frac{1}{m_A} + \frac{1}{m_B} + \hat{n} \cdot (I_A^{-1}(r_A \times \hat{n})) \times r_A + \hat{n} \cdot (I_B^{-1}(r_B \times \hat{n})) \times r_B}$$

$$v_{rel} = \hat{n} \cdot ((v_A + w_A \times r_A) - (v_B + w_B \times r_B))$$

$$j = -(1 + \epsilon) \cdot v_{rel} \cdot m_e$$

Where m_A and m_B are the masses of the body; $\hat{n}$ is the normal vector of the collision contact; r_A and r_B are the positions of the contact points respect to their bodies; v_A, v_B, w_A and w_B are the linear and angular velocity components of the bodies; and ϵ is the restitution coefficient.

The impulse is then applied in the contact point to the body A and the negative value of the same impulse is applied in the body B.

4.2 Constraint Based Solution

The naive approach works but it escalates poorly when the amount of contact points increases. So instead we can use constraints to approximate the result and improve it iteratively [4].

We will approximate an impulse for solving the collision constructing a Jacobian matrix that will be expanded with each constraint. In the case of the contact the only constraints needed are the linear and angular velocities.

That impulse will be accumulated each iteration to approximate the analytical result.

```
for (int i = 0; i < iterations; i++)
{
    for (contact : contacts)
    {
        // save last impulse
        float old_impulse = contact.impulse

        // accumulate impulse
        contact.impulse += compute_impulse(contact)

        // clamp impulse (non negative)
        contact.impulse = max(0, contact.impulse)

        // apply impulse difference to bodies
        vec3 impulse = contact.normal * (contact.impulse - old_impulse)
        contact.body_A.apply_impluse(contact.point,  impulse)
        contact.body_B.apply_impluse(contact.point, -impulse)
    }
}
```

4.2.1 Contact Constraint

The goal of the contact constraint is to limit the velocity of the objects to avoid penetration between them.

$$\dot{C}(x) = JV + b \geq 0$$

To do so, we must solve the Jacobian J that will influence the impulses applied to each of the contact points in the direction of the normal of the contact. In other words, we want to revert the penetration between contact points given as a negative result of

$$d = \hat{n} \cdot (P_B - P_A)$$

Introducing this equation as a position constraint and derivating it we get the terms for the Jacobian

$$C = \hat{n} \cdot (P_B - P_A) \geq 0$$

$$\dot{C} = [v_B + \omega_B \times r_B - (v_A + \omega_A \times r_A)] \cdot \hat{n} \geq 0$$

Using this values, the impulse for the current contact points can be computed as

$$\lambda = -(JM^{-}1J^T)^{-1}(JV + b)$$

where $(JM^{-}1J^T)^{-1}$ is the effective mass as previously described

Applying and accumulating the impulses will end up converging in a solution for the collision where the total energy is conserved. To get a more realistic collision resolution where part of the energy is lost we can add restitution and make the collision be stable using the bias b.

4.2.2 Friction Constraint

Similarly to the collision constraint, friction also limits the velocity of a body based on the contact points. Friction however, will apply impulses in the plane described by the perpendicular vectors $\hat{u}, \hat{v}$, with normal $\hat{n}$ of the contact. The impulses will also be limited to the collision constraint impulse previously computed and the friction coefficient of the bodies.

$$-\mu\lambda_n \leq \lambda_u \leq \mu\lambda_n$$

$$-\mu\lambda_n \leq \lambda_v \leq \mu\lambda_n$$

References

[1] David Baraff. Unconstrained rigid body dynamics. 1997.

[2] Gino Van den Bergen, editor. *Game Physics Pearls*. 2010.

[3] Dirk Gregorius. Robust contact creation for physics simulations. 2013.

[4] Richard Tongue. Solving rigid body contacts. 2012.