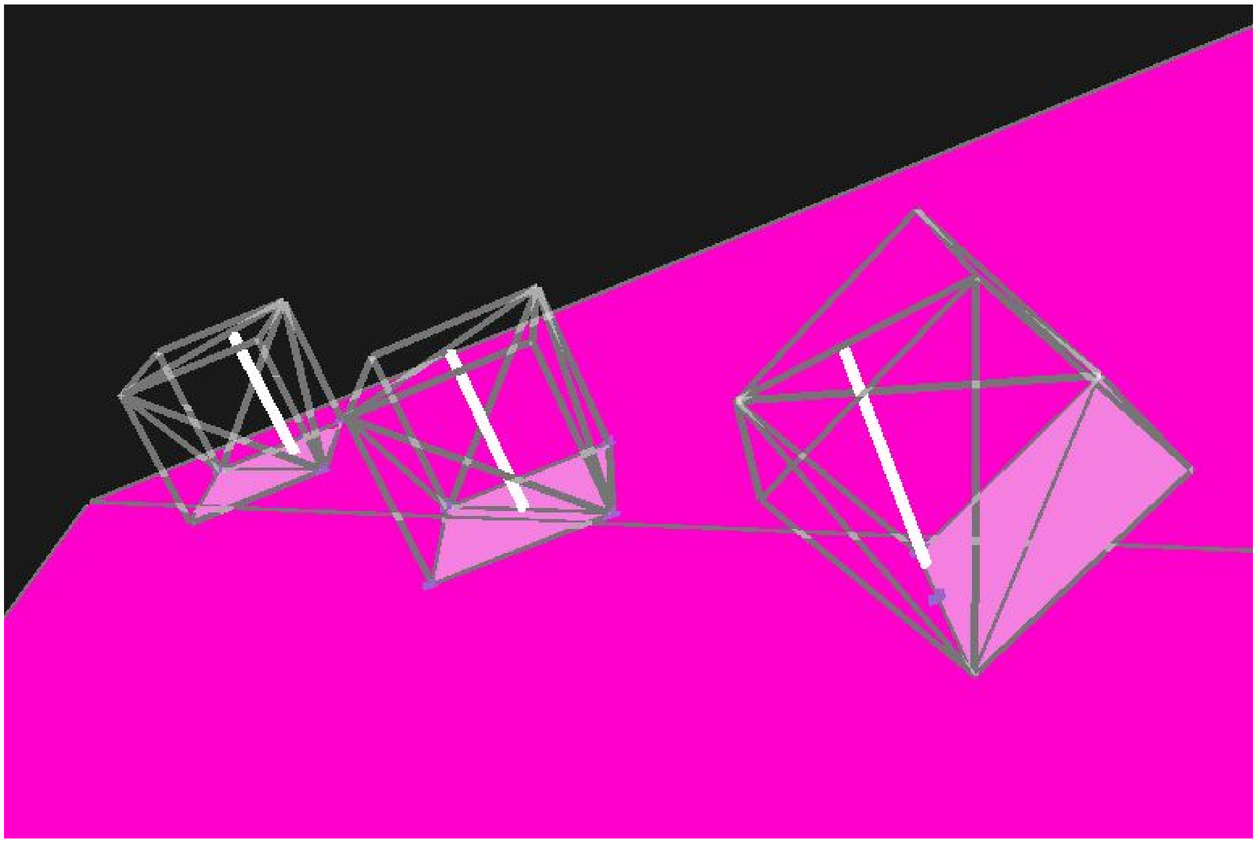
Computer Physics Simulation For Games

Beñat Morisset de Pérdigo

23rd of April 2020, DigiPen Institute of Technology, Bilbao



0. Table of Contents

1. Introduction

This document concludes the course CS 550: Physics Simulation that I have attended at Digipen Bilbao during the semester of spring 2020. During the semester we have learned the basics of 3D physics simulation for games. We have also implemented a small 3D physics engine with collision resolution for any convex shape. This document is an overview of my implementation.

2. Notation & Convention

- 3D vectors will be noted as $\bar{x}$.
- The time derivative will be noted as $\frac{dx}{dt}$
- For the sake of simplicity points will be considered vectors.
- There is only one quaternion: θ
- For all shapes, surface normals will be considered to point outside the shape.
- Points in faces of polyhedra will be in counter-clockwise order.
- The simulation time step will be noted as dt

3. Rigid Body Dynamics

3.1 Properties

The properties define a rigid body. For rigid body simulations they are considered constant in respect to time.

I used the following basic properties:

- Inverse Mass(m):
  As we all have experienced in real life, the mass of an object greatly affects how an object interacts with other objects. This property conditions how forces and impulses are converted into linear motion (the more mass the less a force/impulse will affect the motion).
  I store the inverse mass as it is more handy in many computations.
  By setting this variable to 0 the object will no longer be affected by forces and impulses. I use this to make an object static without the need of storing another variable.

- Inverse Local Volumetric Inertia Tensor(I_{vol}^{-1}): Similar to the mass, but for rotation. The inertia tensor conditions how the torque is converted into local angular motion. It represents how the mass is distributed in space.
  It is in the local space of the body that has the center of mass as origin. For computations in world space, this matrix must be transformed into world space before using it.
  The fact that it is in terms of volume instead of mass is to be able to change the mass without recomputing the whole matrix. I have to multiply by the inverse density each time I want to use it. This is only doable for bodies that have a constant density over their volume.
  Again, I store the inverse because it is the inverse that is used in the integration and so I do not need to invert the matrix each time.

- Volume(V): This property's sole purpose is to be able to convert the volumetric inertia onto the massic inertia. This is how: $I = \frac{m}{V} \cdot I_{vol} \rightarrow I^{-1} = \frac{V}{m} \cdot I_{vol}^{-1}$

The code below shows how I transform the local inverse volumetric inertia into the world inverse massic inertia:

```
mat3 rigidbody::GetInvInertiaWorld() const
{
  if (this->IsStatic()) //checks if this->inv_mass == 0.0f
    return mat3(0);

  const mat3 R = this->GetRotMatrix();
  const mat3 inv_R = glm::transpose(R);

  //Iworld = R * Ilocal * R-1
  const mat3 inv_I = R * this->inv_inertia * inv_R;

  return this->inv_mass * this->volume * inv_I;
}
```

Other sets of basic properties can also be used, like mass and local inertia tensor.

More advanced properties can be added to simulate different behaviors like friction and restitution. I added a friction coefficient to have basic friction simulation.

3.2 State Variables

The state variables are the variables used to describe the motion of a rigid body. Rigid body motion can be described as a composition of pure linear motion of the center of mass and pure angular motion of the body in respect to the center of mass. This simplifies the computations so I chose to use this description.

I used the following state variables:

Linear:	center of mass position: $\overline{x}$	linear velocity: $\overline{v}$	linear force: $\overline{F}$
Angular:	orientation: θ	angular velocity: $\overline{\omega}$	torque: $\overline{\tau}$

By choosing the angular velocity instead of the angular momentum I am already deviating from reality. In my simulation the angular velocity is conserved when no torque is applied, but what should be conserved is the angular momentum, and the angular velocity should be computed from the angular momentum. I chose this deformation of reality because it is not seen as wrong for by humans (unless they specifically hunt for it of course). The goal in a video game is not to replicate perfectly real world physics but to give the illusion of it.

There are other possible choices of variables. Linear and angular acceleration can be used instead of the force and torque for example.

For the orientation I chose a quaternion representation. There exists other representations such as matrix representation, Euler angles, axis and angle. Quaternions have the advantage of being efficient and numerically stable.

3.3 Integration

The relation between the state variable is as follows:

Linear:	$\overline{F} = m \cdot \frac{d\overline{v}}{dt}$	$\overline{v} = \frac{d\overline{x}}{dt}$
Angular:	$\overline{\tau} = I \cdot \frac{d\overline{\omega}}{dt}$	$\overline{\omega} = \frac{d\theta}{dt}$

Therefore, to update the state variables $\overline{x}$, $\overline{v}$, θ, $\overline{\omega}$ we need to integrate. Exact integration is impossible as computer simulations work with discrete time. An approximation must be used. Discrete integration methods are used in such cases: Explicit Euler, Runge-kutta 4 and Semi-implicit Euler to name a few. For this project I have used Semi-implicit Euler as it is simple.

3.3.1 Semi-implicit Euler

In semi-implicit Euler, the first derivative is computed from the last step's second derivative, as in the explicit Euler method. The variable is then computed from the current step's derivative as in the implicit Euler method:

$$x'_{k+1} = x'_k + x''_k \cdot dt \qquad\qquad x_{k+1} = x_k + x'_{k+1} \cdot dt$$

Below you can see my implementation of rigid body motion integration:

```
//linear
lin_velocity += inv_mass * total_force * dt;
position += lin_velocity * dt;
//angular
ang_velocity += InvInertiaWorld * total_torque * dt;
orientation += 0.5f*(quat(0.0f, ang_velocity)*orientation) * dt;
```

3.4 Force, Torque and Impulse

To act upon rigid body motion forces, torque and impulse are often used.

The force is applied on a point. It generates a linear motion change as $\overline{F} = m \cdot \frac{d\overline{v}}{dt}$. It also generates a torque $\overline{\tau} = \overline{r} \times \overline{F}$ where $\overline{r} = \overline{app} - \overline{c.o.m}$.

The torque only affects angular motion: $\overline{\tau} = I \cdot \frac{d\overline{\omega}}{dt}$

Both the force and the torque only take effect if applied for some amount of time. Therefore, I accumulate them each in a variable until they get used at the integration. Then, I reset those two variables to zero vectors once they have been used.

Here you can see my implementation of `AddForce`:

```
void rigidbody::AddForce(const vec3& f, const vec3 world_point)
{
  this->total_force += f;

  const vec3 dr = (world_point - this->pos);
  this->total_torque += cross(dr, f );
}
```

Sometimes we want the effect to be instantaneous, (for solving constraints for example). This is where impulses come in handy. An impulse can be seen as a force that has already been multiplied by the time. This way we affect instantly both the linear and the angular velocity:

$$\overline{F} \cdot dt = m \cdot d\overline{v} \qquad \overline{\tau} \cdot dt = \overline{r} \times (\overline{F} \cdot dt) \qquad \overline{\tau} \cdot dt = I \cdot d\overline{\omega}$$

In my project I started working with only forces and torques, but gradually changed to impulses.

Here you can see my implementation of `ApplyImpulse`

```
void rigidbody::ApplyImpulse(const vec3& impulse, const vec3 world_point)
{
  if (this->IsStatic()) //checks if this->inv_mass == 0.0f
    return;

  //linear
  this->vel += this->inv_mass * impulse;

  //angular
  const vec3 dr = (world_point - this->pos);
  const vec3 impulse_torque = glm::cross(dr, impulse);
  this->ang_vel += this->GetInvInertiaWorld() * impulse_torque;
}
```

4. Inertia Tensor Computation

As the rest of the properties of the rigid body, the inertia tensor must be set before performing the simulation. The difference is that to get believable results this property cannot be set by hand. In fact, the inertia describes how the mass is distributed in space. As I store it in terms of volume, the shape of the object completely determines it.

For well known-shapes such as parallelepiped, cylinder and cone, it is simpler and more accurate to compute it from the explicit formula.

Here you can see my implementation for the full parallelepiped(box) volumetric inertia.

```
mat3 compute_full_box_inertia(const vec3& scale, float* volume)
{
  *volume = scale.x * scale.y * scale.z;

  //square of each scale component
  const float sx2 = scale.x * scale.x;
  const float sy2 = scale.y * scale.y;
  const float sz2 = scale.z * scale.z;

  //diagonal elements
  const float d0 = sy2 + sz2;
  const float d1 = sx2 + sz2;
  const float d2 = sx2 + sy2;

  const mat3 inertia_tensor =
    (1.0f / (12.0f)) *
    mat3
    {
      { d0,  .0f, .0f },
      { .0f,  d1, .0f },
      { .0f, .0f,  d2 }
    };

  return *volume * inertia_tensor;
}
```

To be able to compute the inertia tensor of any convex polyhedra I used Eberly's method. This method computes the volume integrals over the shape one triangle at a time and then sums them all. Alongside the volumetric inertia, it computes the volume and the center of volume. The center of mass is the same as the center of volume as we are only working with bodies that have constant density over their volume. As we want the center of mass to be at the local origin, we must translate the whole mesh to the computed c.o.m. and then either recompute the inertia or apply Steiner's Theorem (Parallel axis theorem).

$$I_{xx}^{B} = I_{xx}^{A} + m \cdot (AB_y^2 + AB_z^2) \qquad I_{xy}^{B} = I_{yx}^{B} = I_{xy}^{A} + m \cdot AB_x \cdot AB_y$$
$$I_{yy}^{B} = I_{yy}^{A} + m \cdot (AB_x^2 + AB_z^2) \qquad I_{yz}^{B} = I_{zy}^{B} = I_{yz}^{A} + m \cdot AB_y \cdot AB_z$$
$$I_{zz}^{B} = I_{zz}^{A} + m \cdot (AB_x^2 + AB_y^2) \qquad I_{zx}^{B} = I_{xz}^{B} = I_{zx}^{A} + m \cdot AB_z \cdot AB_x$$

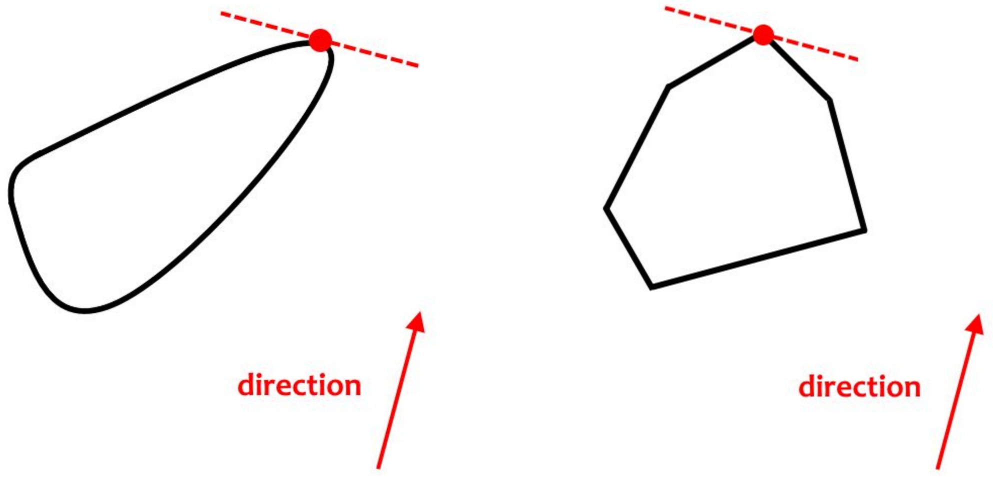
As I work with the volumetric inertia, the mass must be replaced by the volume in the formulas above.

5. Collision Detection

5.1 Support

Before delving into the collision detection algorithms, we need to understand what a support point is, as it is used in both algorithms explained in this section.

The support point of a shape is the point that is most extreme in a given direction. For polyhedra this point is warranted to be a vertex of the shape.



The naive way of searching the support point is doing the dot product of the direction with every vertex of the shape and taking the one that maximizes that value.

5.1.1 Hill-climbing

Hill-climbing is an optimization for the support function for convex polyhedra. It uses the fact that in convex shapes the local maximum is the global maximum. It is a greedy algorithm that works as follows:

1) Pick an arbitrary vertex of the shape as current point
2) Compute its dot product with the direction
3) Compute the dot product with the direction of all connected vertices
4) If none of them is bigger than the current's, we have finished. The current point is the support point
5) Else pick the one that has the highest dot product as current point and go to 2)

The speed of this algorithm greatly depends on the starting point chosen. The closest the starting point is to the support point, the less iterations it has to do.

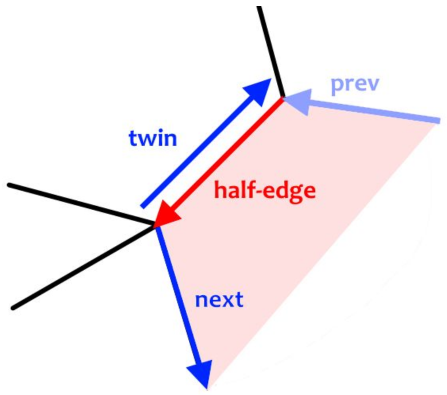
This algorithm poses another challenge: How do we know which vertices are connected to which? Surely not by iterating through all, this would defeat the purpose. The way chosen to be able to know this is storing the mesh as a half-edge mesh.

5.1.1 Half-edge Mesh

It is called like that because it is a mesh composed of directional edges. Each adirectional edge has two directional edges, thus the term half-edge.

Each half-edge has:

- A reference[1] to the vertex at the end of the directional edge
- A reference to the next half-edge in counterclockwise order. Therefore following these pointers we traverse a full face.
- A reference to its twin. The twin is the half-edge that connects the same couple vertices in the opposite direction. This is used to access other faces
- (Optional) A reference to the previous half-edge.
- (Optional) A reference to the face it belongs to. This avoids the need of iterating through the whole face each time we want to get the face.



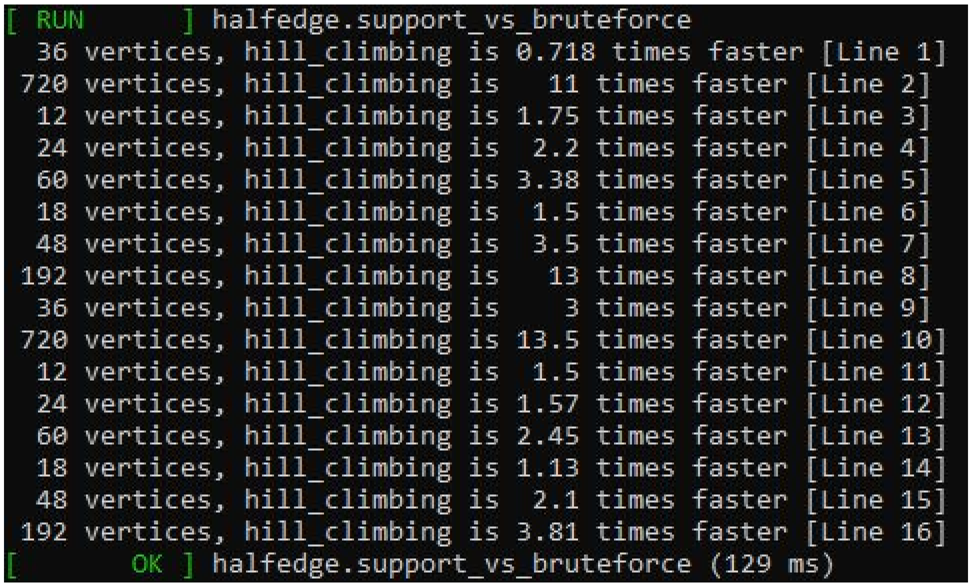
```
struct halfedge
{
  const vec3* vertex;
  halfedge* twin;
  halfedge* next;

  halfedge* prev;
  face* face;
};
```

When generating the half-edge mesh it is a good idea to merge coplanar faces. This further optimizes as it gets rid of unneeded geometry and also prevents problems that may arise when dealing with coplanar faces. Identifying coplanar faces is just a matter of checking if neighboring faces have the same normal.

[1] Reference can be a pointer or an index, or any other way to get access to the element

Below you can see the results I got when benchmarking the naive support function against the hill climbing algorithm using half edges with merged coplanar faces, for two different directions:



```
[ RUN      ] halfedge.support_vs_bruteforce
  36 vertices, hill_climbing is 0.718 times faster [Line 1]
 720 vertices, hill_climbing is    11 times faster [Line 2]
  12 vertices, hill_climbing is  1.75 times faster [Line 3]
  24 vertices, hill_climbing is   2.2 times faster [Line 4]
  60 vertices, hill_climbing is  3.38 times faster [Line 5]
  18 vertices, hill_climbing is   1.5 times faster [Line 6]
  48 vertices, hill_climbing is   3.5 times faster [Line 7]
 192 vertices, hill_climbing is    13 times faster [Line 8]
  36 vertices, hill_climbing is     3 times faster [Line 9]
 720 vertices, hill_climbing is  13.5 times faster [Line 10]
  12 vertices, hill_climbing is   1.5 times faster [Line 11]
  24 vertices, hill_climbing is  1.57 times faster [Line 12]
  60 vertices, hill_climbing is  2.45 times faster [Line 13]
  18 vertices, hill_climbing is  1.13 times faster [Line 14]
  48 vertices, hill_climbing is   2.1 times faster [Line 15]
 192 vertices, hill_climbing is  3.81 times faster [Line 16]
[       OK ] halfedge.support_vs_bruteforce (129 ms)
```

This is by no means a proper performance measurement, but it gives an idea of how both methods compare to each other.

5.2 Separating Axis Theorem

The Separating Axis Theorem (SAT) translates in 3D to the following: two convex shapes do not overlap if and only if at least a plane can be found that separates them. This means that to prove two convex shapes do not overlap it suffices to find a separating plane. Unfortunately, to prove two convex shapes overlap we need to make sure no plane can separate them, and there is an infinity of possible planes.

Thankfully, in computer games we mostly work with polyhedra. For two convex polyhedra no intersection can be proven with a finite amount of planes. The candidate planes are the planes corresponding to each face of each polyhedra, and the planes formed with each edge pair that takes one edge of each polyhedra.

Here is some pseudocode of the overall algorithm:

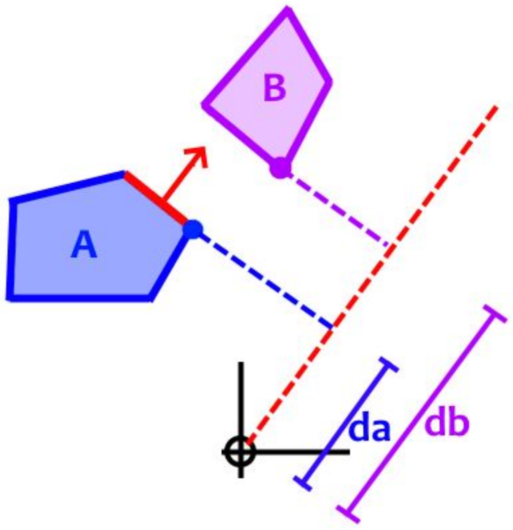
```
bool overlap_SAT(const convex_poly& body_a, const convex_poly& body_b)
{
  //FACES OF A
  for (const face& f : body_a.m_faces)
    return !IsSeparatingPlane(f.get_plane(), body_b);

  //FACES OF B
  for (const face& f : body_b.m_faces)
    return !IsSeparatingPlane(f.get_plane(), body_a);

  //FACES FROM EDGES
  for (const edge& edge_a : body_a.m_edges)
    for (const edge& edge_b : body_b.m_edges)
      return !IsSeparatingPlane(MakePlane(edge_a, edge_b));

  return true;
}
```

To check if a face from a plane is a separating plane I get the support point from the other body in the opposite direction to the normal. Then, I project that point onto the normal and compare it with the projection of any point of the face onto the normal. If the support point has a bigger projection, the plane is a separating plane.



Below you can see the pseudocode for the function that checks if a face plane is a separating plane:

```
bool IsSeparatingPlane(const plane& face_plane, const convex_poly&
other_body)
{
  const vec3 support_pt_b = support_point(-face_plane.normal, other_body);

  const float d_a = dot(pt, face_plane.normal);
  const float d_b = dot(support_pt_b, face_plane.point);

  if (d_b > d_a)
    return true;

  return false;
}
```

For a plane generated by one edge of each body there are a few ways to check if it is a separating plane. I found my own way, which is the following:

1) Compute the normal as the cross product of both axis
2) Find the support point in the direction of the normal, and the negative normal in both shapes
3) Project all these support points onto the normal
4) The pair of projections coming from a shape form and interval. If the intersection of both intervals is the empty set, the normal belongs to the separating plane

```
bool EdgesFormSeparatingPlane(const edge& edge_a, const edge& edge_b,
  const convex_poly& body_a, const convex_poly& body_b)
{
  const vec3& a0 = edge_a.start;
  const vec3& a1 = edge_a.end;
  const vec3& b0 = edge_b.start;
  const vec3& b1 = edge_b.end;

  const vec3 axis = glm::cross(a1 - a0, b1 - b0);

  //cannot build separating plane with parallel edges
  if (equal(axis, vec3(0))) return false;

  //create intervals
  vec2 d_a_pair
  {
    dot(support_point(axis, body_a), axis),
    dot(support_point(-axis, body_a), axis)
  };
  vec2 d_b_pair
  {
    dot(support_point(axis, body_b), axis),
    dot(support_point(-axis, body_b), axis)
  };

  if (intersect_intervals(d_a_pair, d_b_pair) == empty_set)
    return true;

  return false;
}
```

5.3 Gilbert-Johnson-Keerthi

Before explaining the algorithm, I will briefly explain some of the core concepts needed to understand it: Minkowski sum, Minkowski difference and Voronoi region.

Minkowski sum: It is the region of space formed by all the points resulting in the sum of one point of each shape.
Minkowski difference: It is a Minkowski sum in which one shape is negated. One useful property of the Minkowski difference is that its support point in a given direction is equal to the support point in that direction of the first shape, minus the support point in the opposite direction of the other shape:

$$Mink.Diff.Support(A, B, \overline{dir}) = Support(A, \overline{dir}) - Support(B, -\overline{dir})$$

So it is not needed to actually compute the full Minkowski difference to find its support!
Voronoi region: The Voronoi region of a feature is the region of space that is closer to that feature than to any other.

Now onto Gilbert-Johnson-Keerthi's algorithm explanation. This algorithm uses the fact that two convex shapes intersect if and only if their Minkowski difference encloses the origin. So, if a simplex that encloses the origin can be found inside the Minkowski difference, it means that the Minkowski difference encloses the origin and therefore the two shapes overlap.

The algorithm detects that the Minkowski difference does not enclose the origin when a support point of the Minkowski difference in the direction from a simplex of the Minkowski difference towards the origin is still on the same side of the origin as the simplex.

The algorithm is as follows:

1) Pick an arbitrary starting simplex of the Minkowski difference (can be a single point)
2) Check if the simplex encloses the origin.
3) If it does, we are done. The shapes overlap.
4) If it does not, only keep the part of the simplex that has the origin inside its Voronoi region. It may be the entire simplex when the simplex is not a tetrahedron.
5) Make the search direction perpendicular[2] to the remaining simplex that goes towards the origin
6) Find the support point of the Minkowski difference in the search direction
7) If the support point is not on the other side of the origin (dot product is negative), we are done. The shapes do not overlap.
8) Add the point to the simplex and go back to 2)

[2] At this point the simplex is either a point, a segment or a triangle. If it is a point, just take the direction from this point towards the origin.

Here you can see an overview of my implementation:

```
bool overlap_GJK(const convex_poly& body_a, const convex_poly& body_b
  const std::vector<vec3> starting_simplex)
{
  std::vector<vec3> simplex = starting_simplex;

  while (1) //an iteration limit is needed if dealing with curved shapes
  {
    /* returns true if the simplex is a tetrahedron
    that encloses the origin. Else it returns false
    and only keeps the feature of the simplex whose
    Voronoi region contains the origin. It also
    modifies dir to be the search direction for the
    next iteration */
    if (DoSimplex(&simplex, &dir))
      return true;

    P = support_point_minkowski_diff(dir, obj_a.m_head, obj_b.m_head);

    if (glm::dot(P, dir) < 0.0f)
      return false;

    simplex.push_back(P);
  }
}
```

This algorithm the following advantages over SAT:

- More efficient. In SAT, we must check every single candidate plane to prove that there is no overlap, but in JGK the algorithm will determine that the origin is not contained in very few iterations. For contact generation this also applies when there is an overlap, as in SAT, we must go through each candidate plane to find which has the minimum penetration.
- It can be used with curved shapes (a support function for them must be implemented) while SAT only works with polyhedra
- Can be improved a lot by giving it a good starting simplex. A good choice for that is to give it the previous final simplex.

The main disadvantage is that it is more challenging to generate from it the collision data.
In my physics engine, I have chosen to only use it when no collision data is needed.

6. Contact Generation

To be able to simulate collision between rigid bodies, some information about the collision is needed. What contact information is needed depends on the contact resolution. In my case I have the following contact data:

```
struct contact
{
  float penetration; //positive value
  vec3 normal;
  vec3 tangent;

  std::vector<vec3> points;

  //references to both colliding rigid bodies
  rigidbody* A = nullptr;
  rigidbody* B = nullptr;
};
```

6.1 SAT

To be able to generate contact information with SAT we need to keep track of the plane that gave the least penetration, and which features generated it (face or edges). In my implementation the penetration is the length of the interval resulting from the intersection of the two projection intervals (see `EdgesFormSeparatingPlane` in section 5.2).

Once we have found contact plane, the contact normal and the contact tangent are trivial to get:

- The normal is the normal of the contact plane
- The tangent is an edge of the generating face or one of the generating edges

If the contact plane has been generated from two edges, the contact point is simply the closest point between the two edges.
If the contact plane has been generated from a face. The contact points are the points of that face, clipped inside the other shape.

I performed the clipping with Sutherman-Hodgeman's algorithm with the planes taken from each of the polyhedron's faces.

7. Collision Resolution

Now that we know when two convex polyhedra intersect, and we can get the contact information, we can use it to simulate collision.

7.2 Iterative Contact Constraints Solver

This method considers the collision to be a set of constraints. Constraints are formulas that control the behavior to our bodies.

Each iteration the solver solves the constraints for each contact point applying impulses. The solution converges with the iterations.

Below there is the pseudocode for the overall algorithm:

```
for (int i = 0; i < iteration_count; i++)
  for (contact& c : contacts)
    SolveContactConstraint(contact);
```

Below you can see some pseudocode to give an overview of how the each constraint is solved:

```
void SolveContactConstraint(contact& c)
{
  for (int i = 0u; i < c.points; i++)
  {
    const float old_impulse = c.normal_impulses[i];

    //accumulate the normal_impulses
    const float calc_impulse = normal_impulse(c, i);
    c.normal_impulses[i] += calc_impulse;

    //to ensure impulse is not used to keep objects in contact
    c.normal_impulses[i] = glm::max(0.0f, c.normal_impulses[i]);

    const vec3 impulse_vec =
      (c.normal_impulses[i] - old_impulse) * c.normal;

    //what does this do? changes the velocities (v and w)
    c.A->ApplyImpulse(-impulse_vec, c.points[i]);
    c.B->ApplyImpulse(impulse_vec, c.points[i]);


    //proceed similarly for the friction
    //...
  }
}
```

The no-penetration constraint is:

$$J \cdot V \;>=\; 0$$

where

$$J \;=\; \left[-\widehat{n}^T \quad -(\bar{r}_A \times \widehat{n})^T \quad \widehat{n}^T \quad (\bar{r}_B \times \widehat{n})^T\right] \;,$$

$$V \;=\left[\bar{v}_A + (\bar{\omega}_A \times \bar{r}_A) \quad \bar{\omega}_A \quad \bar{v}_B + (\bar{\omega}_B \times \bar{r}_B) \quad \bar{\omega}_B\right]^T$$

The formula can be derived from the position constraint or directly from the velocities.

To improve stability a bias can be added with Baumgarte's Stabilization: $J \cdot V \;+\; b >= \;0$
With $b = \;-\beta \;\cdot \frac{penetration}{dt}$. β is a tuning factor between 0 and 1.

Similar to the no-penetration constraint, I added a friction constraint, which is solved on each iteration after the no-penetration constraint

The contact friction constraint is:

$$J \cdot V \ >= \ 0$$

where

$$\begin{aligned} J_1 &= \begin{bmatrix} -\hat{t}_1^{\,T} & -(\bar{r}_A \times \hat{t}_1)^T & \hat{t}_1^{\,T} & (\bar{r}_B \times \hat{t}_1)^T \end{bmatrix} \quad , \\ J_2 &= \begin{bmatrix} -\hat{t}_2^{\,T} & -(\bar{r}_A \times \hat{t}_2)^T & \hat{t}_2^{\,T} & (\bar{r}_B \times \hat{t}_2)^T \end{bmatrix} \quad , \\ V &= \begin{bmatrix} \bar{v}_A + (\bar{\omega}_A \times \bar{r}_A) & \bar{\omega}_A & \bar{v}_B + (\bar{\omega}_B \times \bar{r}_B) & \bar{\omega}_B \end{bmatrix}^T \end{aligned}$$

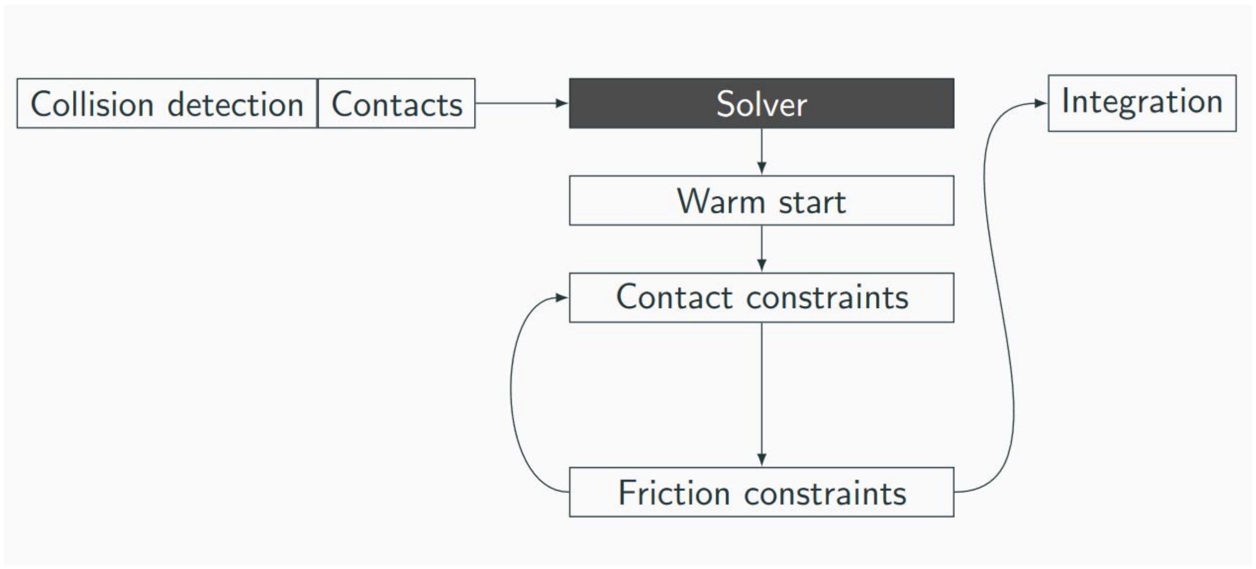
8. Sleeping

Sleeping is a trick used in games to improve stability and avoid objects to be slightly moving when they should be at rest. The idea is to make the object immune to impulses and avoid integration after some time of it being almost at rest.

The way I have implemented it is by having a timer in the rigid body that would be reset at each integration if the angular or the linear velocity's magnitude is bigger than a threshold. Once the timer runs out, the object becomes asleep. I then wake the object upon applying to him a relevant impulse or force, and I also wake all objects colliding with this newly awoken object.

After some tweaking, it has significantly improved the appeal of my demo and increased the number of cubes I can stack.

9. Conclusion



On this paper I have touched on all the core parts of a simple 3D physics engine for rigid body simulations.

Many optimizations could be done, like adding a broad phase and a middle phase to the collision detection using bounding volumes and space-partitioning data structure. Or using GJK instead of SAT, even for contact generation.

Stability has much room for improvement. Slops could be an easy improvement as well as warm start for the constraint solver.

More features could be added, like restitution, general constraints or even fractures.

As a whole I believe this project gave me a solid basis from which I can now grow into a game physics programmer with time and dedication.

10. References

Ming-Lun "Allen" Chou | 周明倫, Game Physics Series:
http://allenchou.net/game-physics-series/

J. Peraire, S. Widnall, Lecture L26 - 3D Rigid Body Dynamics: The Inertia Tensor:
https://ocw.mit.edu/courses/aeronautics-and-astronautics/16-07-dynamics-fall-2009/lecture-notes/MIT16_07F09_Lec26.pdf

David Eberly, Polyhedral Mass Properties (Revisited):
https://www.geometrictools.com/Documentation/PolyhedralMassProperties.pdf

Casey Muratori, Implementing GJK:
https://caseymuratori.com/blog_0003