Simulation of Ragdolls

Iñigo Sena

November 2023

1 Introduction

The purpose of this project is to research, understand and implement the various algorithms, data structures and mathematical foundations behind physics simulations in the constraints of real time (for instance video games or film production); with the goal of implementing a fully functional ragdoll. To achieve this objective, we will research topics such as: the collision pipeline, rigid body representation, numerical methods, acceleration structures and etc.

1.1 Framework

This project will be done entirely by scratch in the C++ programming language except for the use of certain libraries for window management (GLFW), OpenGL extension (GLAD), loading geometry (TinyObj) and math (glm).

The basic framework is built using OpenGL and a basic Bling-Phong shader with a single directional light. Each object shown in screen has its own rigid-body and geometry (mesh and half-edge).

2 Physics System

The Physics System is the part of the program responsible for the dynamics and kinematics of all the objects in the scene; and its the only part of the simulation that can update an object's position and rotation.

2.1 Equations of Motion

The approach we are going to take in order to represent the motion of our objects is based in Newtonian dynamics [Eberly, 2010]; by using the Newton-Euler equations of motion. The idea for this approach is to derive Newton's second law to get an ordinary differential equation to represent the position of an object's center of mass.

$$\sum F = m \cdot a = m \cdot \frac{dv}{dt} = m \cdot \frac{d^2x}{dt^2} \tag{1}$$

From this equation we can get a differential equation for the position $\dot{x} = v$. Furthermore, we can also define an ODE for the linear momentum ($P = mv$) of the object:

$$\frac{dP}{dt} = m\frac{dv}{dt} = ma = F \tag{2}$$

Analogous ODEs for the orientation of a rigibody can be derived. For example, the change in rotation of a rigidbody $\dot{R}$ (not to be confused with the angular velocity ω) is defined as

$$\frac{dR}{dt} = \omega * R \tag{3}$$

Where $*$ represents the skew matrix multiplication. We can also get a differential equation for the angular momentum L

$$\frac{dL}{dt} = \tau \tag{4}$$

All of these differential equations will be used to compute the position and orientation of all the objects in the scene at each frame.

2.2 State Vector and Rigidbody Representation

The state vector of a rigidbody is a conceptual vector that stores both spatial information (position and rotation) and velocity information (linear and angular momentum) [Baraff, 1997]. Thus, let $Y(t)$ be the state vector of a rigidbody:

$$Y(t) = \begin{bmatrix} x(t) \\ R(t) \\ P(t) \\ L(t) \end{bmatrix} \tag{5}$$

Then, we can get the derivative of Y dY/dt from equations (1),(2),(3) and (4) as

$$\frac{dY}{dt} = \begin{bmatrix} v(t) \\ \omega(t) * R(t) \\ F(t) \\ \tau(t) \end{bmatrix} \tag{6}$$

There are still other quantities that we have to compute not discussed above: linear velocity, inertia tensor and angular velocity. These are called derived quantities and in order to compute those, we use a formula derived from either one of the equations above (v and ω for example), or from its own definition (the inertia tensor). The latter is defined in world space, and its computed using a user defined model space inertia tensor (the body inertia).

$$\begin{aligned} v &= \frac{P}{m} \\ I^{-1} &= R \cdot I_{cm}^{-1} \cdot R^T \\ \omega &= I^{-1} \cdot L \end{aligned}$$

Then, the last variables we need are: mass and the inertia of the body. These two variables are defined in object creation and are immutable.

2.3 Numerical Methods

Since we have multiple differential equations which all revolve around force and torque; if we knew how those will be applied at all times we could try to solve them analytically instead of numerically and this problem would be much simpler. However, we can't predict how a force or torque will be applied each frame. Then, we are going to use numerical solvers in order to get the new state vector [Eberly, 2010]. For this project I implemented the explicit Euler method and the sympletic Euler method.

2.3.1 Explitic Euler

Given an initial value problem and a first order ordinary differential equation

$$\begin{aligned} &\frac{dy}{dt} = f(t, y) \\ &y(t_0) = y_0 \end{aligned}$$

We can approximate a solution for the equation by doing the following:

$$y_{n+1} = y_n + h \cdot f(t_n, y_n), \forall n \in \mathbb{N} \tag{7}$$

That is, we start from our initial value and each iteration we increment the previous value by the derivative and a certain step $h \in \mathbb{R}$. In our case, $y_n = Y(t)$, $f(t, y) = dY/dt$ and $h = \Delta t$. Then, equation (7) is:

$$Y(t + \Delta t) = \begin{bmatrix} x(t) \\ R(t) \\ P(t) \\ L(t) \end{bmatrix} + \Delta t \cdot \begin{bmatrix} v(t) \\ \omega(t) * R(t) \\ F(t) \\ \tau(t) \end{bmatrix} \tag{8}$$

2.3.2 Sympletic Euler

The above method works for simple cases like constant force and torque but breaks in more complex cases such as a pendulum with angles slightly bigger than around 10°. This happens because every time the derivative is 0, we are increasing the approximation linearly (even if the function is not linear), giving us incorrect values. Given the system of differential equations:

$$\begin{aligned}
&\frac{dx}{dt} = f(t, v) \\
&\frac{dv}{dt} = g(t, v) \\
&f(t_0) = v_0 \\
&g(t_0) = x_0
\end{aligned}$$

We can approximate a solution by

$$v_{n+1} = v_n + h \cdot g(t_n, x_n) \qquad\qquad x_{n+1} = x_n + h \cdot f(t_n, v_{n+1}) \tag{9}$$

This is really similar to Explicit Euler but one of the variables is dependent on the value computed this iteration instead of the previous. This means that in cases where Explicit Euler would fail ($dx/dt = 0$) because of accumulated error, Sympletic Euler "fixes" itself by having the approximate derivative in this t_n instead of previous t values. For this implementation, we would apply (8) to the derivatives as:

$$\frac{dY}{dt}(t + \Delta t) = \begin{bmatrix} v(t) \\ \omega(t) * R(t) \\ F(t) \\ \tau(t) \end{bmatrix} + \begin{bmatrix} F(t)/m \\ \alpha * \dot{R} \\ 0 \\ 0 \end{bmatrix} \tag{10}$$

Where $\alpha = I_{cm}^{-1} \cdot (\tau - \omega \times (I_{cm} \cdot \omega))$ (this equation can be derived from Euler's equation of motion). Then, now that we have our derivative computed numerically, we apply (8) again but with the new derivative.

3 Narrow Phase Collisions

Now that we have built a system for objects to move; we have to take the first step in order to apply contact forces to objects. To do so, we are going to build a system to detect objects colliding and by how much they are colliding in order to apply a certain force and fix that penetration.

3.1 Minkowski difference

The mathematical principle to determine if two convex shapes intersect is done through the Minkowski difference between two objects A and B (expressed as $A - B$); which is defined as the set of all vectors from any point of B to any point of A [Van Den Bergen, 2003]. We can prove that if we can find the origin in $A - B$, then A and B intersect, therefore, if we can find at least one common point $P \in A \cap B$, that same point in $A - B$ would be the 0 vector. Let $p \in A \cap B$, then $p \in A$ $p \in B$ and $p - p = \vec{0} \in A - B$.

Thus, the main problem we want to solve in collision detection is finding the origin in the Minkowski difference between two objects.

3.2 Half Edge and Hill Climbing

When we start detecting collisions between objects we will want to have what are called **support points**, i.e. points in our object that are the furthest in a certain direction $\vec{v}$. If we took a simple, brute force approach we could get these points in $O(n)$ time complexity [Ericson, 2004]; but we can use certain algorithms and data structures to accelerate this process

3.2.1 Half Edge data structure

The Half Edge data structure is a way of describing object geometry in terms of edges instead of vertices and triangles like we are used to in computer graphics. Half Edges are composed of **edges** and **faces**. And edge consists of:

- **Origin vertex:** Starting position of the edge
- **Face:** Which face this edge is part of
- **Next edge:** The next edge this edge is connected to (similar to how linked list have a *next* node)
- **Twin edge:** An adjacent edge to this one. They connect the same vertices but with opposite direction and are part of different faces

While a face consists of:

- **Normal:** The normal of the face pointing outwards
- **Edge:** One of the edges of this face.

If the Half Edge was properly built, we should be able to loop around a face with a single edge. Moreover, we are able to get to any vertex from any starting point.

This data structure is created after loading new geometry since it is based on the vertices of the models. At first, we are going to build it with the same edges and faces as our mesh. Then, we are going to identify twin edges. Lastly

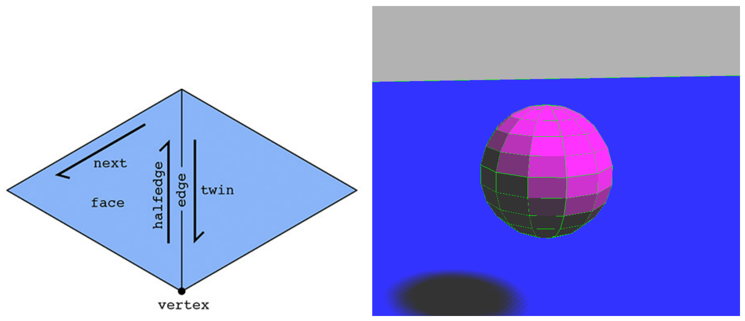

Figure 1: On the left: Example of the HalfEdge data structure. Source: University of California, Berkeley. On the right: screenshot of a triangular mesh with a Half Edge defined (the green lines represent the edges).

we are going to identify which faces can be merged. Two faces can me merged if one of their edge's twin is an edge on an adjacent face and if they share the same normal. Figure 1 shows an example of how two triangles could be represented as a Half Edge.

3.2.2 Hill Climbing

Hill Climbing is an algorithm which takes the bruteforce approach of going through all vertices and, with the support of the Half Edge data structure, it *greedily* explores vertices in order to find the furthest point alongside a direction.

The algorithm consists of taking a starting vertex V_0 and checking all outgoing directions (i.e, taking all outgoing edges from this direction and checking which is furthest). Then, we are going to take the edge wich is furthest in a direction v, and take the endpoint V_1. Then, we are going to iterate until we find a vertex V_i which is furthest in a certain direction. There is, however, a caveat to this algorithm: it only supports convex meshes. Nevertheless, for this project we are going to assume all our geometry will be convex.

3.3 Separating Axis Theorem

In order to detect when two objects are colliding we are going to use the **Separating Axis Theorem**, which consists of trying to find certain axes that separate two objects to determine if they are *not* colliding.

Let O_1 and O_2 be two objects we are trying to know if they intersect. To determine if two objects are colliding, we are going to iterate over the faces of O_1: $f \in \mathcal{F}$. Then, we are going to find a support point in the direction of $-\vec{n}$ (the normal vector of our face f with opposite direction). To avoid unnecessary model transformations, we can apply the model matrix of O_1 and the inverse model matrix of O_2 to get the direction we are looking for in O_2's model space.

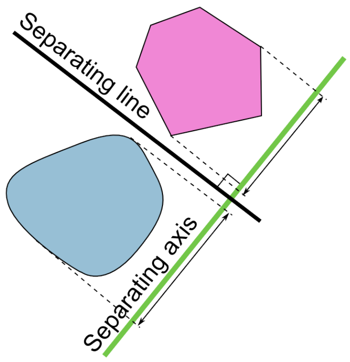

Figure 2: Example of SAT in 2D. Source: Wikimedia commons

Once we find this point P, we create the plane defined by f in world space. We then compute the **Signed Distance Function** of our point with the plane; and if this distance is positive (i.e, the point is "above" the plane), we know that $\vec{n}$ is a separating axis and the objects won't collide. If the distance is negative, repeat this process with all faces.

This method works because we are implicitly computing the faces of the Minkowski difference between O_1 and O_2 and checking if the origin is there. There is still an issue present. If we see Minkowski differences computed explicitly, we can see that some faces are not defined by the subtraction of our faces, but also defined by the parallelogram created with four vertices (figure 3) , two from O_1 and the other two from O_2 [Choi et al., 2005]. Thus, we also need to take edges into consideration when computing the intersection between two objects.

The naïve method to check edge vs edge intersection consists of going through both objects' edges, computing the cross product $e_1 \times e_2$ and using them as a candidate for a separating axis. This means the complexity of our algorithm would become $O(n^3)$. We can improve this complexity by using Gauss Maps and duality transforms.

3.3.1 Gauss Maps and Duality Transforms

A **Gauss map** is an alternative representation of our shape represented in the unit sphere $\mathbb{S}^2$, where the normal vectors of our faces become points in the sphere, and edges become **arcs** in our sphere. Implementation wise this transformation is done implicitly rather than explicitly computing the Gauss map and storing it in memory.

In the context of our algorithm, we will loop over the edges of both objects and check if the arc of the Gauss Maps intersects. To do this, we define a, b, c, d

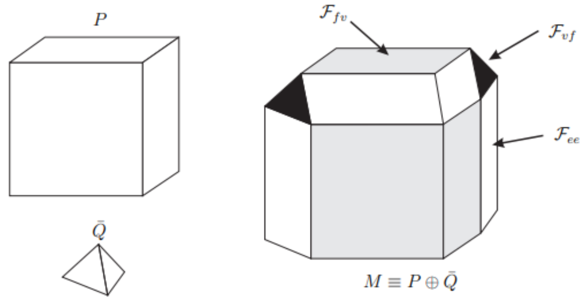

Figure 3: The right shape shows the Minkowski difference between P and Q. The face labeled as $\mathcal{F}_{ee}$ is generated by the cross product between edges. Source: *Collision detection of convex polyhedra based on duality transformation* Department of Computer Science, The University of Hong Kong

as the normals of the faces adjacent to both edges (a and b of e_1 and c and d of e_2) [Choi et al., 2005]. Now, we compute the Scalar Triple Product of these values and we get check the following condition:

$$
\begin{gathered}
[c \cdot (b \times a)] \cdot [d \cdot (b \times a)] < 0 \\
[a \cdot (d \times c)] \cdot [b \cdot (d \times c)] < 0 \\
[a \cdot (c \times b)] \cdot [d \cdot (c \times b)] > 0
\end{gathered}
$$

If all three are true, the edges are a possible candidate, we compute the cross product between them and then use the resulting vector to do the same computations we did with faces. If any of them is false, it means the arcs don't intersect and these two edges are not a possible candidate

3.4 Contact Information

Now that we know if two shapes are in contact we need to generate certain information in order to separate both objects. To do that, we need certain information which will be stored in a structure called a **Contact Manifold**. The contact manifold will stores:

- **Minimum Translation Vector**: The minimum distance between the two bodies
- **Feature**: If the contact was a face or edge contact; and if it was a face, did we use O_1's faces or O_2's
- **Contact Points**: Which points from either object penetrate the other object. These points also contain extra information such as penetration and a scalar λ, which will be used later for constraints.

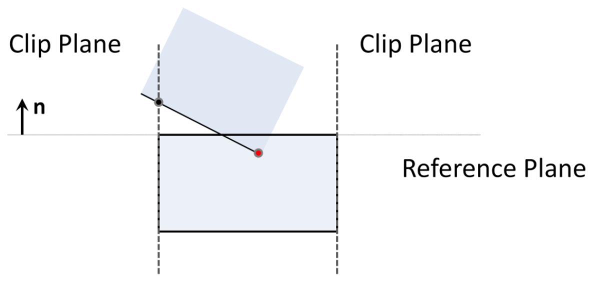

Figure 4: Example of which features are present in a Face vs Face scenario in 2D. Source: Games Developer Conference and Valve Corporation.

Both the feature and the minimum translation vector can be generated on the fly while computing SAT. The problem comes with the contact points of the objects. If the feature is an edge vs edge, we just take the two closest points in either edge. For face vs face, we need to do more work

3.4.1 Face vs Face contact point generation

The first step in contact point generation is to keep track of the **reference face**, that is, the face whose normal will be the closest to the other object. From this, we need to search for the **incident face**, the most *anti-parallel* face of the other object. To do so, we compute the distance between the reference face and the other object's face's and get the closest one. [Gregorius, 2015]

Now that we have both the incident and the reference face, we need to get the points. To do so, we take all the points forming the incident face and *clip* them against the adjacent faces of the reference face using the Sutherland-Hodgman algorithm. Then after getting all the points we get rid of the points above the reference face and we found all contact points.

4 Collision Resolution

Now that we have generated the contact information, we need a way to apply certain forces in order to correct the position of our penetrating objects and resolve the collision between them. There are multiple approaches to solve the problem, but we are going to apply the **impulse** based method.

4.1 Impulse

An impulse J is defined as a force applied during time step $\Delta t = t_2 - t_1$. That is:

$$J = \int_{t_0}^{t_1} F(\tau)\, d\tau$$

The impulse can also be reinterpreted as the change of momentum $\Delta\vec{P}$ from t_0 to t_1. In reality what we are going to do is apply a force F in the time step of our simulation Δt. Thus, given the vector from the objects center of mass to the collision point $r = (p_{cm} - p_{collision})$, a normal vector $\vec{n}$ and the impulse J we can update velocities and momenta (linear and angular) as:

$$\begin{aligned}
\vec{v_2} &= \vec{v_1} + \frac{J}{m} \\
\vec{\omega_2} &= \vec{\omega_1} + J \cdot I^{-1} \cdot \vec{r} \times \vec{n} \\
\vec{P_2} &= \vec{P_1} + J \cdot \vec{n} \\
\vec{L_2} &= \vec{L_1} + J \cdot \vec{r} \times \vec{n}
\end{aligned}$$

Here, $\vec{r}$ and $\vec{n}$ are both known, the only problem is computing the impulse. We could apply Eberly's formula to get J [Tamis, 2015] if we had a single contact point. The issue comes when we have multiple contact points. We can compute J for the first point P_1, but the problem is that we modified the state of the object. We could compute the impulse at P_2 without updating the body or we could update the body and then compute P_2's impulse. The solution is to do the former per each point. If we consider multiple simultaneous contact points, we can write our problem as [Eberly, 2010]:

$$\begin{cases}
\text{Minimize: } ||\mathbf{Af} + \mathbf{b}||^{\mathbf{2}}, \text{ Subject to:} \\
\mathbf{f} \geq 0 \\
\mathbf{Af} + \mathbf{b} \geq 0
\end{cases} \tag{11}$$

This is a Convex Quadratic Programming problem; which can be translated into a Linear Complementary Problem (LCP) and solved with algorithms such as Lemke's algorithm. This, however, is not optimal for real time applications[1]. Instead, we are going to treat contacts as **constraints**.

4.2 Constraints

A constraint is a concept in mechanics in which the degrees of freedom of at least one object can be limited by a mathematical function $C(p) = 0$ [Chappuis, 2013]. In physics programming, they are often compared to shaders in computer graphics [Catto, 2014], as they are the fundamental building blocks of physics engines. As with shaders, there are certain constraints that are most common (contact,

[1]This is not true if Lemke's algorithm is implemented in the GPU as it is suitable for parallelization [Lauritzen, 2007]

friction, etc.); but they also allow for creativity in making unique interactions.

For most constraints, the initial definition will be given by a **position constraint**; that is, the function $C(p)$ limits how an object might move/rotate. Nevertheless, the end goal of our constraint is finding an impulse; therefore we will work with $\dot{C}(p)$. We can derive that $\dot{C}(p) = J \cdot \vec{v} + b$ [Catto, 2009], where J is defined as the **Jacobian** of the function $C(p)$ and b (called the **bias** or **push factor**) is a constant used for multiple purposes (Baumgarte stabilization,motor constraints, restitution...) [Catto, 2009]. Then, for the constraint to be satisfied we want to make $\dot{C}(p)$ *as close to 0* as possible. Thus, we are going to define our *corrective impulse* $P_c = J^T \cdot \lambda$ as the impulse that will modify the objects velocity to minimize our velocity and position constraint functions.

In order to compute λ, we can derive a formula from Newton's second law in impulse form $\mathbf{M\Delta v} = \mathbf{P_c}$ (where $\mathbf{M}$ is the mass matrix, i.e, a square matrix where the diagonal is either the identity times the object's mass or the object's inertia tensor), the definition of our constraint $\dot{C}(p) = 0$ and the definition of our impulse $P_c = J^T \cdot \lambda$ [Tamis, 2015].Then, we have the following equations:

$$\begin{aligned} \mathbf{v_{post}} &= \mathbf{v_{prev}} + \mathbf{M^{-1}P_c} \\ \mathbf{P_c} &= \mathbf{J^T}\lambda \\ \mathbf{J^T v_{post}} &= 0 \end{aligned}$$

Therefore, from these equations we can get the following formula for λ

$$\lambda = \frac{\mathbf{Jv_{prev}} + b}{-\mathbf{JM^{-1}J^T}} \tag{12}$$

4.3 Contact Constraint

Now that we have a general definition for a constraint, creating the contact constraint is a matter of defining $C(p)$, $\dot{C}(p)$ and J. A contact constraint is different from the previous constraint in which the condition for it to me bet is not $C(p) = 0$, but rather $C(p) \geq 0$. Let P_a and P_b be the contact points of objects A and B; and $\vec{r_a}$ and $\vec{r_b}$ the vector from their center of mass' position to the contact point. Then $C(p)$ and $\dot{C}(p)$ can be defined as: [Chappuis, 2013]

$$C(p) = (P_a - P_b) \cdot \vec{n} \tag{13}$$

$$\dot{C}(p) = \begin{pmatrix} -\vec{n}^T & -(\vec{r_a} \times \vec{n})^T & \vec{n}^T & (\vec{r_b} \times \vec{n})^T \end{pmatrix} \begin{pmatrix} \vec{v_a} \\ \vec{\omega_a} \\ \vec{v_b} \\ \vec{\omega_b} \end{pmatrix} \tag{14}$$

From this we can see that the vector multiplying the velocities is the Jacobian of the constraint. In contact constraints we are going to use the term $b = \frac{d}{\Delta t}$ (where

d is the penetration of the point) for stabilization (this is called Baumgarte stabilization), and, if we wanted to add restitution we could also add ϵJv to b (where $\epsilon \in [0,1]$). With this we have modeled a constraint for a single contact point, but if we want to have multiple contact points we reach the same problem of having an LCP. However, we can now resolve constraints by using iterative solvers.

4.4 Multiple contacts, iterative solvers

We have defined what a constraint is and how to model contacts as a constraint; now we are only missing how to deal with multiple contact points at the same time. In order to compute the impulse for the object to be in a correct position, we are going to use **iterative solvers**; i.e. we are going to *iteratively* go over all contact points and compute λ. The algorithm we are going to use is called **Projected Gauss-Seidel**. This algorithm is a numerical method used to iteratively approximate systems of linear equations, that is, problems that look like: $\mathbf{Ax} = \mathbf{b}$. In this case, $\mathbf{A} = \mathbf{JM^{-1}J^{T}}$, $\mathbf{x} = (\lambda_1, \lambda_2, \ldots \lambda_n)$ and $\mathbf{b} = \frac{1}{\Delta t} v - \mathbf{J}(\frac{1}{\Delta t}\mathbf{v} + \mathbf{M^{-1}F_{external}})$ (where v is a vector filled with the biases of the constraints and $\mathbf{v}$ is the velocity) [Tamis, 2015].

The algorithm begins by selecting a starting λ_s, which can be any number but we can also use the result of the previous frame (warm start) [Catto, 2009]. Then, we are going to iterate i times through the contact points (where i is a user defined constant). Next, we are going to compute λ for this iteration and accumulate it into the contact point $\lambda_{cummulative} += \lambda_{current}$ and clamp $\lambda_{cummulative}$ to be greater or equal to 0 (this is the *projected* part of Gauss-Seidel). We then compute $\Delta\lambda$ with the previous accumulated λ and the current accumulated λ (so if we overshoot the correction it can also move the object towards the other object), and update the velocities and momenta of the object, but not the position. After doing all iterations, we will have an approximated velocity to correct the penetration and we can do a single Euler step to correct the object's position and rotation.

5 Ragdolls

With all of these tools at our disposal, we can now start seeing how to implement ragdolls. In order to do that, we are going to define ragdolls as multiple rigidbodies connected with **joint constraints**.

5.1 Ball-and-Socket Joints

The ball joint constraint, also known as ball-and-socket joint, is a constraint in which the three degrees of translational freedom are limited, i.e, the objects can rotate freely between themselves but can't move from an *anchor point.* Then, let P be the anchor point in world space coordinates, we then store p_{mi} the anchor point in *model space* of both objects; and at each frame, we compute p_{wi} the anchor point transformed with the current model to world matrix of each object. Then, let r_i be the vector from an objects position to p_{wi}. The constraint is defined as: [Cline, 2002]

$$C(p) = p_{w1} - p_{w2} \tag{15}$$

$$\mathbf{J} = \begin{pmatrix} E_3 & r_1^* & -E_3 & -r_2^* \end{pmatrix} \tag{16}$$

And with these equations we can get a formula for λ; the main difference being that the Jacobian is now a 3x12 matrix instead of a 1x12 matrix. Therefore λ will be a vector of three values, one per each axis.

5.2 Hinge Joints

A hinge joint is similar to a ball joint; it limits the three translational degrees of freedom; and it also limits two degrees of rotational freedom; only letting the objects rotate along a certain axis. This constraint has two parts: the translation constraint and the rotational constraint. The translation part is the same as the ball-and-socket joint. The rotational constraint works by giving an axis of rotation a, then storing it in the model space of each object (a_1 and a_2). Then, at each frame we compute b_2 and c_2 two perpendicular vectors to a_2. Thus, our constraint is defined as: [Chappuis, 2013]:

$$C(p) = \begin{pmatrix} a_1 \cdot b_2 \\ a_1 \cdot c_2 \end{pmatrix} \tag{17}$$

$$\mathbf{J} = \begin{pmatrix} 0 & -(b_2 \times a_1) & 0 & (b_2 \times a_1) \\ 0 & -(c_2 \times a_1) & 0 & (c_2 \times a_1) \end{pmatrix} \tag{18}$$

5.3 Building the Ragdoll

Now that we have the necessary joints; we need to create certain body parts and connect them by joints. The starting point will be the chest; which will have five ball joints attached: two for each upper arm, two for each thigh and one for the head. Then: each upper arm will have a hinge joint attached to the forearm with the axis being the Y axis (elbow); and each thigh will have a hinge connected to the lower leg with the X axis (knee) as a rotational axis.

6 GPUs in physics programming

All of the computations proposed so far are expected to be done in the CPU. One question that comes to mind is if this process can be accelerated by leveraging the parallelization powers and the higher FLOP count of a GPGPU; or removing instruction overhead by using SIMD commands. The latter is more straightforward as we just need to be careful about alignment and, every time we want to do computations (such as an Euler step, or modify an impulse in a constraint) we could do it in a single step. For GPGPUs, the process is more complicated.

6.1 Interfacing with the GPU

The main way of interfacing with the GPU to do physics programming is by using a GPGPU API such as OpenCL or CUDA. With this, we can dispatch *kernels* for each different systems of our program [Coumans, 2013]. The problem then becomes synchronization and memory transfers. In the case of game physics this can become a huge overhead as, if we want to modify rigidbodies by game logic, that logic can only be executed in the CPU. Nevertheless, if we only want to render a simulation without any kind of user input or external logic, this could be a valid approach. Some games also use this to render and simulate geometry that does not affect gameplay (i.e, rubble in an explosion, particles, etc.); which by could be made so they live exclusively in the GPU; removing work from the CPU.

6.2 Shape Representation in the GPU

In this project we've used the Half Edge data structure to represent geometry for physics. In the GPU, specially for more complex simulations, we can represent it as a group of particles [Bell et al., 2005]. In this method, we define a uniform three dimensional grid around the object. We then raytrace our object, and if a ray has an odd number of intersections with our shape, it means we are inside the rigidbody. Now, no matter if our shape is convex or concave, our collision detection algorithm has changed to finding if two particles are colliding (a simple sphere vs sphere test). Moreover, the generation of this shape can be accelerated using a technique called *depth peeling* [Lauritzen, 2007].

6.3 LCP in the GPU

As mentioned before, we can compute a global solution of impulses by using Lemke's algorithm; which is more accurate than our Projected Gauss-Seidel solver: but its also more expensive. Using the GPU, we can dispatch a kernel for each row of our matrix M, and a compute grid for each collision pair. According to NVIDIA and assuming 2007 hardware, there is a 328% increase in queries per second using a LCP algorithm in CPU vs GPU [Lauritzen, 2007]

7 Conclusion

This project has proved to be more difficult in some aspects, and easier in other. I thought that some algorithms such as SAT would be harder to implement, but they were easier than I thought they would; and things such as integrating proved to be harder since there wasn't too much information about Semi Implicit Euler. One of the biggest challenges so far has been floating point precision; and how small changes up to 0.0001 can, after a certain amount of frames, make a huge difference in the simulation and break it completely. One other thing that has proved to be quite difficult is not *how* to implement things, but *where*. For example, I could find a lot of information about how to solve a contact constraint with n points (which is difficult by itself); but it didn't say *where* to implement it and how to coordinate it with the rest of the systems.

One of the things I've learned is why every time something is abused in a game (specially older games), it is usually the physics system. They are extremely feeble and can break easily if you know where to exploit them. This is unsurprising considering the whole system is built in approximations and assumptions.

References

[Baraff, 1997] Baraff, D. (1997). An introduction to physically based modeling: rigid body simulation ii—nonpenetration constraints. *SIGGRAPH course notes*, pages D31–D68.

[Bell et al., 2005] Bell, N., Yu, Y., and Mucha, P. J. (2005). Particle-based simulation of granular materials. In *Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 77–86.

[Catto, 2009] Catto, E. (2009). Modeling and solving constraints.

[Catto, 2014] Catto, E. (2014). Understanding constraints.

[Chappuis, 2013] Chappuis, D. (2013). Constraints derivation for rigid body simulation in 3d.

[Choi et al., 2005] Choi, Y.-K., Li, X., Wang, W., and Cameron, S. (2005). Collision detection of convex polyhedra based on duality transformation. *The University of Hong Kong*.

[Cline, 2002] Cline, M. B. (2002). *Rigid body simulation with contact and constraints*. PhD thesis, University of British Columbia.

[Coumans, 2013] Coumans, E. (2013). Gpu rigid body simulation using opencl.

[Eberly, 2010] Eberly, D. H. (2010). *Game physics*. CRC Press.

[Ericson, 2004] Ericson, C. (2004). *Real-time collision detection*. Crc Press.

[Gregorius, 2015] Gregorius, D. (2015). Robust contact creation for physics simulation.

[Lauritzen, 2007] Lauritzen, A. (2007). *GPU Gems 3*. Addison-Wesley.

[Tamis, 2015] Tamis, M. (2015). 3d constraint derivations for impulse solvers. *Retreived Form: http://www.mft-spirit.nl/files/MTamis_Constraints. pdf.*

[Van Den Bergen, 2003] Van Den Bergen, G. (2003). *Collision detection in interactive 3D environments*. CRC Press.