

Tasks

Check in code:

- ☒ Are we really using the [Jacobi method rather than the Gauss-Seidel method](#)? (or ask Matt).

Discuss with Matt:

- ☐ Would the solution for the bounce-too-soon problem i describe [here](#) work? If so, why aren't we using it?

Tech Design: [Physics Core - Physics World](#)

World API:

- multiple objects → PhysicsWorld methods
- single objects → Physics...Handle methods

All API is

- **non-blocking** (🤔🌀 calls return immediately, not waiting e.g. for a simulation update),
- **multi-threaded** (can be called from different threads).

Objects are referenced by **handles** (lightweight pass-by-value objects, kinda like shared_ptr).

See [Tech Guide: Physics Objects](#).

Physics update:

- **Normal Update**
 - **destroys handles** scheduled for destruction since previous Normal Update
 - **simulate physics** for given timestep
 - evaluates **persistent queries**
- **NoSolve Update** (when physics is disabled or game is paused)
 - **only destroys** handles scheduled for destruction since previous Normal Update
 - (no physics, no queries)
- **Correction Update** (when applying correction to local networked players)
 - simulate **only the corrected bodies** for given timestep
 - (no queries)

Tech Guide: [Physics Objects](#)

In *italics*, things i'm guessing or deducing, to be verified.

Body:

- **physical** properties and state (inertia, kinematics)
- has a **transform** that maps to an entity or bone
- *has **shapes**, each shape with a **relative transform***
- see [Tech Guide: Physics Rigid Bodies](#)

Constraint:

- **limits** the motion between **two bodies**, one of which can be a static body (e.g. the world)
- relates to the two bodies through **relative transforms**
- can be disabled
- ☐ see refs

Contact:

- position constraint between **points** on **two shape** instances
- available in collision reports and contact modifiers
- ☐ see refs

Collision report:

- info about all the collisions during the update
- ☐ see refs

Shape:

- *references* (i.e., doesn't own, so they can be shared) a **convex hull**, **heightfield**, or **mesh data**
- attached to a **body** with a **relative transform**
- has a radius around the convex hull
 - ☐ what does this mean?
- can be disabled
- has a material index for looking up collision material properties
- stores collision filtering info so that collision and queries can be done on subsets of shapes
- no easy means to access all shapes of a body at this level
 - ☐ what does this mean?
- see [Tech Design: Shapes](#)

Heightfield:

- **geometry of a shape** representing a **terrain**, based on a regular grid; may include holes

Convex hull:

- **geometry of a shape** representing a **convex hull** with points and edges (as a **half-edge mesh**)
 - in a half-edge mesh, we have vertices, edges, and faces, but also half-edges
 - each half-edge $v_1 \rightarrow v_2$ contains references to
 - its twin $v_2 \rightarrow v_1$
 - the next half-edge CCW around the face
 - v_1
 - its associated edge
 - its associated face
 - it's a good balance between simplicity and geometry processing capability
- can represent simple shapes with few vertices: sphere (1), capsule (2), triangle (3), box (8)
- **collision meshes** are represented by multiple convex hulls
 - *only convex hulls and heightfields can collide; meshes can only be used in queries*

Mesh:

- **geometry of a shape** representing a **triangle mesh**
- used only in queries, **not collisions**

Query:

- **cast against shapes**
- can use collision filtering (see [Tech Design: Collision Filtering](#)) to filter shapes
- ☐ see refs

Query result:

- references the shape that is hit
- additional info about the part of the shape
 - ☐ what does this mean?

Tech Guide: [Physics Rigid Bodies](#)

Created via FBPhysicsWorld. Controlled through PhysicsBodyHandle.

Typically, an FBPhysicsComponent owns a body and associates it with a ComponentEntity (or part of it if multiple bodies are associated with the same entity). By default, behaviour is independent of the entity type (StaticModelEntity, DynamicModelEntity, Banger, etc).

See [Tech Guide: Body Transforms and Spaces](#).

Motion types:

- dynamic

- **kinematics simulated** each frame (see [Tech Guide: Physics Rigid Body Solver](#))
 - subject to **constraints**
 - subject to **contact constraints** generated by collisions between its shapes and other shapes
 - **automatically update** the position of its associated **entity** (or part of the entity)
- keyframed
 - **kinematics updated** by the solver to achieve a **desired target**
 - **not** affected by **constraints** or **collisions**
 - body **velocity** set **automatically** before the update so it moves to the target
- static (authored; no body)
 - **shapes** with **fixed position**
 - **no need** to create a runtime **body** for fixed shapes ⇒ optimisations
- static (runtime)
 - a dynamic or keyframed body can be made **static temporarily** so it doesn't move
- debris (obsolete) → now low-quality dynamic body

Quality:

- high
 - **self-colliding**, **many** solver iterations, **high-quality** collision
 - only affected by other **high-quality** bodies
- medium
 - **self-colliding**, **few** solver iterations, **low-quality** collision
 - only affected by **high-** and **medium-quality** bodies
- low
 - **non-self-colliding**, **few** solver iterations, **low-quality** collision
 - only affected by **high-quality** bodies

The specific settings for each quality (number of solver iterations, number of partitions, the qualities that affect it, etc) can be configured in game.cfg or in code

Sleep:

- a dynamic or keyframed body can be **put to sleep** when at rest
- a sleeping body is temporarily **not simulated**
- a sleeping body **wakes up automatically** when there's a chance they may **interact** with another body
- a sleeping body can also be **woken up explicitly**
- **drives wake up automatically** involved bodies when they change (e.g. parameter change)
 - ☐ why not other kinds of constraints?

Tech Design: [Physics Core - Bodies](#)

PhysicsBodyHandle provides access to the dynamic simulation of a [rigid body](#).

Possible states:

- **dynamic** (simulated under constrained motion, affected by collisions, joints, and drives)
- **keyframed** (updated according to their velocities only)
- **static** (don't move)

There's a special static body: the **world body**. Static shapes and constraints are attached to it.

The FBPhysicsWorld API features methods to

- access the **world body**
- **create** a body
- **schedule destroy** for a body

PhysicsBodyHandle provides

- get and set current 2nd-order kinematic state (position, orientation, and linear and angular velocity and acceleration)
 - ☐ i wonder what setting the acceleration does
- get and set motion type
- get and set enable/disable
- get and set mass properties
- changing other simulation properties

When an entity is scaled, the body **mass properties are scaled at runtime**.

Tech Design: [Physics Core - Object Handles & Lifetime](#)

Tech Guide: [Body Transforms and Spaces](#)

"Body space" is what i call "local frame".

"Mass space" (or "mass frame") is a frame aligned with the principal axes of inertia (cross terms of the inertia tensor are null in this frame).

Internally, the solver computes the body space transform **B**, and computes the mass frame transform **M** from **B** and the relative body transform **R** (the three are related by $\mathbf{B} = \mathbf{R} \cdot \mathbf{M}$)

- ☐ really? i would have guessed the solver computes **M** rather than **B**

What the user sees is normally $\mathbf{C} = \mathbf{R}^{-1}$. If the user changes the value of $\mathbf{C}$, then $\mathbf{R}$ and $\mathbf{M}$ are automatically updated from $\mathbf{B}$ and $\mathbf{C}$.

Tech Guide: [Physics Rigid Body Solver](#)

(Converted from [Solver Core](#). Apparently identical, except the next-to-last image is missing in Confluence, and links haven't been updated to point to other Confluence pages.)

The solver simulates a system of rigid bodies under **Newton's laws**, subject to **additional constraints** (including **collisions**). During a “**simulation step**” (or “solver step”) the state of the system goes to the next simulated time. The solver supports **variable time step**.

The additional constraints are handled by introducing the minimal force and torque (*and impulse and torque impulse?*) possible that keeps the constraint satisfied.

A “local” solver isn't enough when we have more than one constraint; we need a “global” solver:

1. **start** with a global solution made up of the **local solutions** for each constraint
2. **iterate** (**improve** the global solution by solving **each constraint locally in turns** using the actions generated by the **current** global solution) for a number of times

The iteration over constraints is done a number of times. The global solution tends to **converge**. The method is equivalent to a projected [Gauss-Seidel](#). This process is sped up by **constraint compilation** (an alternative representation optimised for repeated access).

After all the iterated constraint solves comes the **dynamic update**: the equations of motion are solved using external forces and constraint forces. Then, the **spies update** (spies selectively gather info about constraint forces and deliver it through something like notifications).

☐ are spies still a thing?

Details:

- **mass ratio and stability**: an interaction between a heavy body and a light body is a rigid system; avoid it or allow for a sufficient number of iterations
- **parallel processing**: because of the iterations over individual constraints, we can only parallelise through **islanding** (across sets of bodies where bodies in one set don't interact with bodies in a different set)
- **variable timestep issues**: (this is what i already now: different timesteps lead to different solutions with different degrees of accuracy, leading, for too high values, to **discrete timestep issues** where a body can go from one side of a wall to the other side without hitting the wall)
- **unsolvable configurations** will inject energy into the system and possibly diverge
- **small violations, stabilising effect of contact caching**: constraints can be cached and reused over a number of simulation frames when the two bodies involved haven't moved much from the original situation

- **inertia and solver convergence:** in some situations, increasing inertias can lead to better stability; an example is ragdolls: increasing the rotational inertia of elements doesn't affect much the overall inertia of the ragdoll, but can make joints work better

EAPhysics: [Contacts](#)

I don't know if the information in this wiki page has been transferred to Confluence.

Contact constraints prevent bodies from **interpenetrating**. The solver's job is to **determine the forces** to achieve this.

Interpenetration is computed with **separation distance**, which becomes 0 at contact, and negative when there is interpenetration. The constraint is therefore

$$\text{separation distance} \geq 0$$

When interpenetration isn't just at a single **contact point**, a **contact patch** is computed, and its **vertices** are used instead.

$$\text{reaction force} = \text{normal force} + \text{friction force} + \text{restitution force}$$

Normal force acts to prevent interpenetration with no work. It's normal to the surface that's penetrated.

Restitution "defines how much momentum is transferred by impact" (i would have said it defines how much kinetic energy is conserved; what isn't is transformed into some other kind: sound, heat, etc): 0 is total absorption (clay), 1 is total conservation (bouncy objects).

Restitution force is **normal** to the surface that's penetrated. Apparently, restitution is defined per body, but should be defined per contact (i.e., per pair of bodies).

Friction force prevents sliding. It's in the surface that's penetrated. **Static** friction happens for no sliding, and is limited by *static friction coefficient* $\times$ *normal force*. Dynamic friction happens when sliding, and is *dynamic friction coefficient* $\times$ *normal force*. I believe when we mention "normal force" here, we mean "*normal force + restitution force*".

There's a **contact frame**, aligned with the surface that's penetrated, so that

- **X** is the contact normal,
- **Y** is aligned with the relative velocity of the bodies at the contact point in the contact tangent plane,
- **Z** is $X \times Y$.

Solving one contact probably causes another one to fail. Instead of solving all contacts at once, they're **solved in turns over multiple iterations** (i think contacts and constraints are solved together; contacts *generate* constraints):

for n iterations
 for c in *contacts*
 solve c

Contact generation uses **geometry** to generate **contact constraints**. Generated contact constraints are **potential**: if they do generate forces, they become **real**. In order to anticipate the need to apply forces to avoid interpenetration, **padding** is added. Contact padding can depend on velocity (growingly thicker in the velocity direction). Contact generation can be customised. A contact constraint is defined by

- connectivity: pointers to the two bodies involved;
- geometry: contact normal, contact position on each body and their relative velocity;
- and physics: static friction, dynamic friction, restitution coefficient, masses of the two bodies (for mass biasing, i.e. pretending the involved bodies have different masses just for the contact).

Position correction: If in spite of efforts in the first phase (*displacement correction*), two objects **interpenetrate**, the solver **repositions** them in a second phase, trying (but slightly failing) not to inject energy into the system. See [EAPhysics: How Contacts Work](#).

Contact caching reuses contact constraints from the previous frame and avoids some geometry recomputation as long as the involved objects don't move much. **Contact manifolds** describe the conditions for the cached contacts to still be valid.

Tricky cases:

- the **bullet-through-paper problem**: not even detecting a fast moving objects flies through another one; extending the volume of contact detection can solve this, but it may also generate invisible “force walls”
- **impossible situations** (such as a fixed-moving-fixed object sandwich where the moving body is penetrating both fixed bodies) can generate extremely large forces and instabilities.

EAPhysics: [How Contacts Work](#)

Stages:

1. contact generation (query pairs of collision volumes and generate contact constraints)
2. contact compilation (computes Jacobian and determines thresholds for position correction)
3. solve (determine forces and apply position correction)
4. dynamic update (EOM integration)

Contact compilation:

- compute the *current position difference* $\Delta \mathbf{x}^t = \mathbf{C} \left(\mathbf{x}_B^t - \mathbf{x}_A^t \right)$, where $\mathbf{C}$ is the contact frame

- compute the *projected position difference* $\Delta \mathbf{x}^{t+1} = \Delta \mathbf{x}^t + \mathbf{C} (\Delta \mathbf{d}^t + \Delta \mathbf{s}^{t+1})$, where $\Delta \mathbf{d}$ and $\Delta \mathbf{s}$ are the displacements due to current velocity and acceleration (probably something $\mathbf{v} \Delta t$ and $\frac{1}{2} \mathbf{a} \Delta t^2$)
- compute present penetration $\Delta \mathbf{x}^t \cdot \mathbf{n}$ and projected penetration $\Delta \mathbf{x}^{t+1} \cdot \mathbf{n}$, where $\mathbf{n}$ is the contact normal; using their signs and the value of the restitution coefficient e , there are several cases
 - $\Delta \mathbf{x}^t \cdot \mathbf{n} < 0$, $\Delta \mathbf{x}^{t+1} \cdot \mathbf{n} > 0$, $e > 0$ (not penetrating, going to penetrate, some restitution)
 - $\Delta \mathbf{x}^t \cdot \mathbf{n} < 0$, $\Delta \mathbf{x}^{t+1} \cdot \mathbf{n} > 0$, $e > 0$ (not penetrating, going to penetrate, no restitution)
 - $\Delta \mathbf{x}^t \cdot \mathbf{n} > 0$, $\Delta \mathbf{x}^{t+1} \cdot \mathbf{n} > 0$, $e > 0$ (currently penetrating)

for each of these cases, a displacement limit $\mathbf{l}_{d,x}$ is computed; i don't quite understand how it works

- ☐ understand how the Jacobian is computed
- ☐ understand how the displacement limit works

There's a [broken link](#) to a doc about the Jacobian, called "Solver Contacts and Joints". Maybe it's the same contents as in [Tech Design: General Contact Constraints](#).

EAPhysics: [Solver Contact Constraint](#)

Tech Design: [General Contact Constraints](#)

Tech Design: [Physics Core - Contact Constraints](#)

A contact constraint is broken into

- **non penetration** constraint (1D along normal), which ensures
 - contact points don't cross the tangent plane,
 - their final relative velocity is the initial relative velocity times minus the restitution coefficient
- **friction constraint** (2D perpendicular to normal)
 - contact points stay coincident but the required impulse is limited by normal impulse times static friction, or dynamic friction if static friction is exceeded

The problem is **linearised** for efficiency but also so that a rolling ball doesn't sink into the ground when rotating around the contact point..

The mechanisation is thus:

$$\mathbf{v}' = \mathbf{v} + \mathbf{g} \Delta t + \sum \mathbf{j}/m$$

$$\mathbf{x}' = \mathbf{x} + \mathbf{v}' \Delta t$$

where the primes are values after numerical integration, and $\mathbf{j}$ are impulses, methinks.

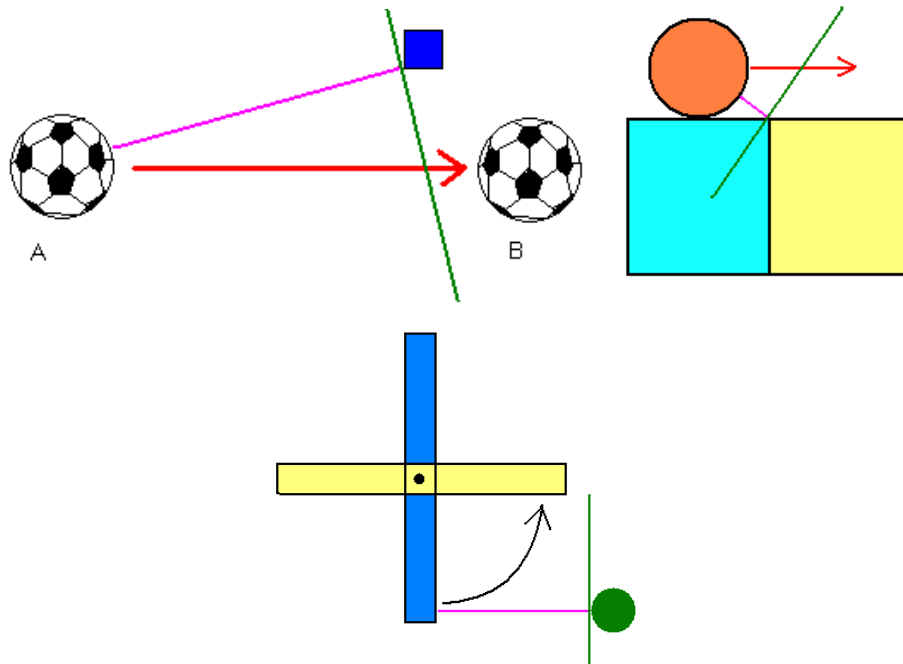
With the added $\mathbf{j}$ due to constraints, we can solve either for $\mathbf{v}$ (for restitution) or for $\mathbf{x}$ (to satisfy the non-penetration constraint). We solve for $\mathbf{v}$, which causes the “[bounce too soon](#)” problem: the collision appears to happen at the beginning of the frame, when the impulse is applied, not at the correct moment and position in the frame (when the bodies actually collide).

💡 Wouldn't the following solve the “bounce too soon” problem?

Instead of doing $\mathbf{x}' = \mathbf{x} + \mathbf{v}' \Delta t$, compute the time of collision $\Delta t_c = \frac{\Delta \mathbf{x}^t \cdot \mathbf{n}}{(\Delta \mathbf{x}^t - \Delta \mathbf{x}^{t+1}) \cdot \mathbf{n}}$ clamped to $[0, 1]$, and do $\mathbf{x}' = \mathbf{x} + \mathbf{v} \Delta t_c + \mathbf{v}' (\Delta t - \Delta t_c)$.

EAPhysics: [Goal Post Problem](#)

If a body is moving fast, the contact computed at the beginning of a frame may be completely inapplicable at the end, which would cause ghost rebounds (this wouldn't happen if we re-evaluated the contact at the end of the frame):



The doc mentions [trajectory lookahead](#) solves this problem, but i didn't see how, unless it refers precisely to re-evaluating the contact constraint at the end of the frame.

Tech Design: [Physics](#)

<http://5ety7tpkim5me6eszuwcje7bmy25pbtrjtue7zkqggziljwqy3rrikqd.onion/FpXU> - Solver Maths

Displacement Based Rigid body Dynamics - Tom Waterson 2020

Summary of Displacement_Based_Rigid_Body_Dynamics.pdf, which i didn't study in depth.

Derives equations for constrained motion from the principle of least reaction, using Lagrange multipliers. Mentions the Gauss-Seidel method for solving the system. Lays out the general algorithm, and shows how to use projected Gauss-Seidel for inequality constraints (i.e., contact constraints, i think). Applications to ball-in-socket and angular-axle constraints. Shows how to introduce non-linear potential terms, using linear spring and damper (which have quadratic potential) as examples (this is used for ragdolls).

Physics Solver Maths - Tom Waterson 2018

Summary of solvern timer.pdf, which i didn't study in depth.

The EOMs are integrated using 1st-order explicit (accel to vel) and implicit Euler (vel to pos). Derives constraints from a Taylor expansion of the constraint conditions, which involves the Lagrangian, and by minimising an energy term $\Delta x \cdot M \Delta x$, thus obtaining the Lagrange multipliers. Describes the pseudo-inverse formulation and the Tikhonov regularisation (adding an extra damping term) to stabilise the solution near singularities.

Gauss-Seidel to solve. Some possible modifications to the solver algorithm: high-fidelity constraints (more expensive but more accurate), relaxation (for Gauss-Seidel stability), explicit Jacobian (re-computing the Jacobian for each iteration for $t + 1$, instead of reusing its value for t), warm starting (use the final value from the previous frame as an initial value for this frame), inequality constraints (projected Gauss-Seidel), limiting the solver impulse (clamping forces and torques), position correction (for contacts, we want to control both velocity and position, to a position correction term is added). Using the XPBD method, elastic energies can be added to the solver.

Application to constraints: locked linear, box, cone, drive, contact (including restitution and friction).

Inverse kinematics: constrain the transform of the end of a chain of joints to a target value (also uses Jacobian and Lagrange multipliers).

The Physics Stasis Solver - Tom Waterson 2008

Summary of *stasis_solver.pdf*, which i didn't study in depth.

The solver is position-based: instead of velocity, *displacement*, and instead of force, *force displacement* (displacement change), from which velocity and acceleration can be obtained if needed as

$$\mathbf{v} = \mathbf{d}/\Delta t$$

$$\mathbf{a} = \mathbf{s}/\Delta t^2$$

Displacement and force displacement:

$$\mathbf{d}^{t+1} = \mathbf{x}^{t+1} - \mathbf{x}^t$$

$$\mathbf{s}^{t+1} = \mathbf{d}^{t+1} - \mathbf{d}^t$$

therefore the dynamic update of is

$$\mathbf{d}^{t+1} = \mathbf{d}^t + \mathbf{s}^{t+1}$$

$$\mathbf{x}^{t+1} = \mathbf{x}^t + \mathbf{d}^{t+1} = \mathbf{x}^t + \mathbf{d}^t + \mathbf{s}^{t+1}$$

Constrained dynamics is derived from the Lagrangian. For linear constraint functions $\mathbf{c}(\mathbf{x}) = D\mathbf{x} - \mathbf{k}$, we get additional unknowns $\mathbf{r}$ (the force displacements for the reaction forces) and additional terms in the Jacobian, which are precisely equal to D . Constraint reactions are introduced thus:

$$\mathbf{d}^{t+1} = \mathbf{d}^t + \mathbf{s}^{t+1} + \mathbf{r}^{t+1}$$

$$\mathbf{x}^{t+1} = \mathbf{x}^t + \mathbf{d}^{t+1} = \mathbf{x}^t + \mathbf{d}^t + \mathbf{s}^{t+1} + \mathbf{r}^{t+1}$$

Gauss-Seidel (which requires the diagonal of A not to have any zeros) is used to solve $A\mathbf{x} = \mathbf{b}$:

1. start with tentative solution $\mathbf{x}^{(0)} = 0$ (for instance)
2. for as many iterations as needed, keep improving the solution:

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + D^{-1}(\mathbf{b} - A\mathbf{x}^{(k)}) \text{ where } D \text{ consists of the diagonal terms in } A$$

(which amounts to solving for each x_i using only equation i)

~~(Actually, Wikipedia describes a different [Gauss-Seidel method](#), where $L_*\mathbf{x}^{(k+1)} = \mathbf{b} - U\mathbf{x}$, using the lower triangular L_* and strictly upper triangular U decomposition of $A = L_* + U$. The version described here seems to be a similar but different, less computationally demanding method, that is described as the [Jacobi method](#). But maybe the fact that constraints are solved in turns rather than simultaneously makes the *implemented* method Gauss-Seidel rather than Jacobi.)~~ I now (20250110) think the method described by Wikipedia is the same as ours.

Optimisations: sparse Jacobian, and $\mathbf{r}$ differentially using the Lagrange multipliers.

Over- and under-relaxation for equation solver: $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \mu D^{-1} (\mathbf{b} - A \mathbf{x}^{(k)})$, where $\mu < 1$ stabilises but slows down convergence, and $\mu > 1$ speeds up convergence at the cost of increased instability.

Joint constraints: application to joints with limits (constraint equations are inequalities): linear and angular joint, locking axes (swing cone, swing axle, swing hinge, swing locked, twist arc, twist locked).

Drive constraints: linear and angular; soft drives are like springs with damping, hard drives drive to a target value with damping; limited force.

Contact constraints: resting contact, restitution, zero restitution, friction, position correction.

Tech Design: Physics Core - [Solver Maths Primer](#)

A round-up of maths needed to understand the solver.

Vectors and matrices:

- ⚠ the order of multiplication in fbmath is the opposite from maths; SIMD has the same order as maths
- skew matrix: $\mathbf{b} = \mathbf{r} \times \mathbf{a} = \text{skew}(\mathbf{r}) \mathbf{a}$ where $\text{skew}(\mathbf{r})$ is a matrix

Properties of rigid body motion: nothing i didn't know.

Tech Design: [Physics Core - Understanding Solver Maths using Newtonian Mechanics](#)

(This is Matt's doc explaining the solver without Lagrangian mechanics, adapted from a [similar presentation](#).)

An **impulse** $\mathbf{p}$ is an *instant* exchange of momentum; for rigid bodies, it's an *instant change in velocity* $\Delta \mathbf{v} = \frac{1}{m} \mathbf{p}$.

Relative motion:

$$\Delta \mathbf{v}_1 = \frac{1}{m_1} \mathbf{p}$$

$$\Delta \mathbf{v}_2 = -\frac{1}{m_2} \mathbf{p}$$

$$\Rightarrow \Delta \mathbf{v}_r = \Delta \mathbf{v}_1 - \Delta \mathbf{v}_2 = \left(\frac{1}{m_1} + \frac{1}{m_2} \right) \mathbf{p} = \frac{1}{m_{red}} \mathbf{p}$$

where m_{red} is the reduced mass

In the solver,

velocity is the displacement added to position each step,

acceleration is the displacement added to velocity each step,

impulse is mass times velocity,

force is mass times acceleration.

Integration scheme (for both linear and angular motion):

$$\mathbf{v} \rightarrow \mathbf{v}' = \mathbf{v} + \mathbf{a} = \mathbf{v} + \Delta \mathbf{v}$$

$$\mathbf{x} \rightarrow \mathbf{x}' = \mathbf{x} + \mathbf{v}' = \mathbf{x} + \mathbf{v} + \Delta \mathbf{v}$$

For a regular torque τ ,

$$\frac{d\omega}{dt} = \mathbf{I}^{-1}((\mathbf{I}\omega) \times \omega) + \mathbf{I}^{-1}\tau$$

where, apparently, we're not applying the $((\mathbf{I}\omega) \times \omega)$ bit correctly. For an impulse $\mathbf{J}$, that term disappears (it isn't an instantly applied change), so

$$\Delta\omega = \mathbf{I}^{-1}\mathbf{J}$$

Solving 0th-order constraints:

- problem
 - bodies and constraints between pairs of bodies
 - known initial position and velocity of all bodies
 - $\Rightarrow$ find $\Delta\mathbf{v}$ and $\Delta\omega$ for all bodies that **satisfy all constraints** while **conserving momentum and energy**
 - this might include loss to {friction, restitution, damping} and gain to {compressed spring}
- solution
 - iterate through **{all constraints} {multiple times}**
 - for each constraint, compute the **error** with **current body displacements**
 - find an **impulse** to **null the error** and apply it to the bodies, **updating their displacement**
 - the impulse conserves momentum ($\mathbf{p}_A + \mathbf{p}_B = 0$) and *for point constraints between bodies with zero relative velocity* ($\mathbf{v}_A = \mathbf{v}_B$) the energy transfer cancels out ($\mathbf{p}_A \cdot \mathbf{v}_A + \mathbf{p}_B \cdot \mathbf{v}_B = 0$)

The whole process is similar to solving a linearised system with the Gauss-Seidel method, but with some additional non-linear error computation and impulse clamping.

Procedure to determine the impulse for a constraint:

- describe the next value of (scalar or vector) **error** in terms of current position and orientation and next velocity and angular velocity
- describe next linear and angular **velocities** as **functions** of force and torque **impulses**
- **plug** the latter into the former, equal the next value of error to **zero**, and **solve for impulses**

I believe if we apply this to a **scalar error**, such as for a contact (interpenetration), we have to add **additional conditions** on the impulse; for instance, for a **contact** constraint, the impulse is a **force perpendicular** to the surface.

Tech Design: Physics Core - [Simulating Stiff Springs](#)

Springs are used for drives, where spring and damping can go from zero (no effect) to ∞ (hard constraint, but there can still be a force limit).

$ma = -kx - cv$, which we transform using $s = k/m$ (for spring) and $d = c/m$ (for damping) into

$$a = -sx - dv$$

Parameters s and d are used so we can tune for behaviour, **independent of mass**.

We can use that equation for weak springs with the solver's Euler scheme, but for stiff springs, we'd inject energy and things would explode. Instead, we go to an **implicit formulation**, where acceleration uses **next** values of x and v :

$v' = v + (a + a_e) \Delta t$, where a_e are accelerations from external forces (including other constraints)

$$x' = x + v' \Delta t = x + v \Delta t + (a + a_e) \Delta t^2$$

$$V' = V + A + A_e, \text{ where } V = v \Delta t \text{ and } A = a \Delta t^2$$

$$x' = x + V' = x + V + A + A_e$$

$$A = -Sx' - DV' \text{ where } S = s \Delta t^2 \text{ and } D = d \Delta t^2$$

$$\text{therefore } A = -S(x + V + A + A_e) - D(V + A + A_e)$$

and, solving, $A = -\beta_s x - \beta_d V - \beta_d A_e$, where

$$\beta_s = \frac{S}{1+S+D} \text{ and } \beta_d = \frac{S+D}{1+S+D}, \text{ both in the range } [0, 1]$$

0 → **no spring** (β_s) **or damping** (β_d)

1 → **no displacement** (β_s) **or velocity** (β_d); for $\beta_s = \beta_d = 1$, we get a **hard constraint**

we use **different values** for β_s and β_d , to avoid (but why?) the fact that s affects β_d :

$$\beta_s = \frac{s}{1+s} \text{ and } \beta_d = \frac{s+d}{1+s}$$

but the situation is complicated:

- in code, we're using this β_d for A_e , but should actually use the old β_d or a new tunable parameter
- the predicted critical damping value is off (naturally!)

A general problem with the s and d approach: tuning is good for a single driven body pair, but for a **chain of bodies**, the effective mass is higher, so we need to compensate.

A different perspective: a proposal for a different tuning procedure, where we specify **frequency and decay** directly, and compute β_s and β_d from them; more natural for tuning, but still the same problem with chains of bodies.

Tech Design: [Joints](#)

Tech Design: [Physics Core - Shapes](#)

Two kinds of shapes for collisions: **heightfield** and **convex hull**. Other kinds (the doc is confusing!):

- **aggregate**: read-only collection of convex hulls sharing shape properties (material, radius, collision presets, etc), with an optional spatial index
- **mesh**: contrary to convex hull, can be concave; mesh-vs-mesh and mesh-vs-heightfield haven't been implemented, which effectively limits them to **static or keyframed shapes**

The FBPhysicsWorld API features methods to

- **create** convex-hull and mesh shapes (attached to a given **body**)
- **create** heightfield, convex-hull, and mesh **fixed** shapes associated with a given fixed **partition**)
- create a sphere or a capsule shape (attached to a given body)
- **schedule destroy** for a given shape

Shape capabilities:

- non-fixed shapes can be dynamically attached to a different body
- non-fixed shapes can be moved into different collision layers
- shapes can be dynamically assigned a different material index
- shapes can be assigned to multiple representations of the same object (e.g. collision LODs)
- shapes can collide with a specific representation using collision culling categories
- multiple shapes can share the same heightfield, convex-hull, or mesh data

Shapes are used through a shape handle. The shape handle API allows to

- get and set enable/disable
- get shape type and an appropriately-typed pointer to its data
- get a handle to the body the shape is attached to
- get or set translation and orientation relative to its body
- get or set radius (fatness for collision)
- get or set uniform scale (not for heightfields)

Tech Design: [Collision Materials](#)

Tech Design: [Collision Filtering](#)

SIMD

Tech Design: [SIMD Package](#), [Glossary of SIMD Terms](#)

From Wikipedia's [Single instruction, multiple data](#): data level **parallelism**, but **not concurrency**; each unit runs the **same instructions on multiple data**.

The SIMD package implements the SIMD **processor-independent API** for each supported processor.

Concepts:

- operations done on a vector (with processor-dependent size) of elements in registers
- data is always in registers:
 - only local variables and function arguments passed by value
 - no concept of data address
 - register values are written to streams for other computations
- all access to memory is through stream operations (with processor-dependent layout)

Definitions:

- **elements** are heterogeneous collections of **attributes** (often one attribute per element) with homogeneous **components** (same type: all integer or all float)
 - example of element: capsule (with attributes length, radius, centroid, axis)
 - example of attribute: centroid, which is a Vector3 with components x, y, and z
- **stream**: memory representation of homogeneous (same type, same meaning) elements for different objects, that are atomically accessed
 - example: `jobArguments->Bodies.RelativeBodyPosition`
 - in the compile and solve functions, streams are read into registers using the `<stream>::Gather()` method
- **stream group**: collection of streams whose corresponding elements are part of the same object
 - example: `jobArguments->Bodies`
- **stream region**: contiguous range of elements for which all objects have the same streams defined
 - what is this?
- **scalar**: a single attribute, component, or element for a single body
- **vector**: a number of scalars; this is what SIMD works with in parallel
- **block**: memory layout of a group of consecutive elements in a stream
 - what is this?

There's a cool example of SIMD that computes the Mandelbrot set.

SIMD Types

Vector types: define a register representation of SIMD_NUM_ELEMENTS elements of the given type; only **local variables** or **read-from/write-to** memory using stream load/store operations:

- Bool
- Int
- Float<L, M, T, Q>
- Float2<...>, Float3<...>, Float4<...> (math vectors)
- FFloat33<...>, Float44<...> (math matrices)
 - Float33ξ<...> (ξ: I identity, A antisymmetric, D diagonal, S symmetric)
- Float4Q<...> (unit-norm quaternions)

Stream types: define a register representation of typed stream pointer; only **local variables** or **function arguments passed by value**:

- Int{8,16}Pointer, UInt{8,16,32,64}Pointer, UInt
- FloatPointer<...>
- PackedFloat3Pointer<...> RandomFloat3Pointer<...>, SequentialFloat3Pointer<...>

SIMD In-Register Processing Operations

A list of functions. Common themes:

- never throw; for instance, $\text{ACos}(a)$ clamps its argument to $[-1, +1]$
- conditional functions; for instance, $\text{AddIf}(a, b, c)$ is $c ? a+b : a$
- matrix and vector maths

Review the remaining sections...

... but i couldn't find any immediate usage for them, since the most SIMD i may get is not with streams, where i should be able to get away with just copying existing code, but rather with physics and maths.

Other docs

Check under [Physics Core - Simulation Data](#)

Tech Design: [Physics Core - Reference Solver](#)

A version of the constraint compile/solve code that

- uses a **scalar** (as in non-SIMD-vector) implementation → readable, accessible, natural (can use branches), easy to debug, easy to validate, easy to try new things on
- can be **used interchangeably** with the SIMD implementation:
 - same input from and output to job arguments constraint data
 - works with all examples and tests modulo small numerical differences
- merges all contact types into a **single parameterised constraint** → easy to see differences
- uses common utils and helpers
- can be used for **new** and **custom** (game-specific) **constraints**
- can be used to check the **understanding** of the solver code

Work branch:

- not up to date with dev-na
- virtual stream overlay

Changes:

- Examples/Example: globally enable or disable reference solver
- FBPhysicsWorld: create and enable or disable reference solver
- Private/World/ReferenceSolver.Public/EAPhysics/World/ReferenceSolver:
 - ReferenceSolver: runs reference solver on batches of constraints
 - ConstraintUtils: common maths
 - BodyConstraintHelper: common physics
 - Constraints:
 - ContactConstraint: merged one-for-all contact (except debris)

- DebrisContactConstraint: (untested) debris contact
- DriveConstraint: directional drive
- JointConstraint: linear and angular joints

Check [20240115 \[task\] changed region queries](#) for additional findings.

☐ transfer notes from that doc to here

Tech Guide: [Entity System](#)

The entity system's purpose is to **hook different systems** into the engine and help them **communicate** with each other.

Authoring:

- **.ddf** ⇒ generates C++ and C# classes
- the **runtime** Entity is coded in **C++**
- **macros** in the class generate an **EntityCreator** and hook it up to the **EntityFactory**
- at **spawn**, entity data is sent to the **EntityFactory**, which finds the right EntityCreator to create the Entity instance

Most classes don't inherit from Entity but from **EntityWithBusAndData**, which stores a const reference to the GameObjectData used to create the entity, and a SmartRef to the EntityBus used for communication.

Init order:

1. constructor
2. onCreate()
3. onInit()

Deinit order:

1. schedule destroy (place in a queue to be destroyed at the end of the frame)
2. onDeinit()
3. onDestroy()
4. destructor

Entity owner:

Tech Guide: [Ownership and lifetime](#)

EntityOwner: a **container of entities**. EntityOwner **owns** its entities. EntityOwner forms a tree, where a node is **refcounted by its children EntityOwner and EntityBus**, and **deleted** only when its refcount reaches 0. **Entities don't refcount** EntityOwner. When a EntityOwner is **destroyed**, it **won't destroy owned entities** (but it will assert if it has any left).

EntityOwner::destroyOwnedEntities() destroys its entities, but not its children. Class hierarchy (from [Entity Owner](#)):

- EntityOwner
- SimpleEntityOwner (adds init, deinit, created, and destroy functions, and additional protected interface to customise behaviour; uses the global entityWorld object to delete the entity)
- SubLevel (holds the RootEntityBus and can be used to traverse all children and their siblings within the bus)

EntityBus: refcounted by Entity; lives in and refcounts EntityOwner.

Entity: lives in and refcounts EntityBus; not refcounted (must be explicitly **scheduled for destruction** from code).

Entity creation: always goes through its corresponding realm EntityFactory; this sets up EntityBus but won't necessarily create an entity (e.g., ghosted entities for replication, unspawning/respawning due to culling).

Tech Guide: [Creating entity types - data and runtime](#)

This is *the guide* to coding entities: .ddf files, realm, events, ghosts, networkable, pipeline, runtime...

EntityData:

- .ddf → C++ pipeline/runtime version and C# FrostEd version
- DDF keyword "entity" instead of "class"
- "[ToolboxItem(Editor="...", Path="...")]" makes it available for drag and drop
- the "Realm Realm" property controls the realm; it can be static ("[StaticProperty]") or normal
-

Tech Guide: [Network - How-to](#)

Tech Guide: [Network - How to create your first networked entity](#)

Tech Guide: [Frostbite Physics Resimulator](#)

The resimulator takes a VDB capture and allows the simulation input to be modified to see how it affects the output.

Build with

```
fb nant EAPhysicsCore win64 gensln opensln \  
-- -D:visualdebugger_base.enableio=true
```

(i actually added `-D:visualdebugger_base.enableio=true` to my regular `fb nant EAPhysicsCore win64 gensln buildsln` command).

Run (i found this on my own, so there may be a better way; the doc wasn't specific) by launching

`TnT/Local/Build/EAPhysicsCore/flattened/pc64-vc-final/bin/example/EAPhysics.Resimulator.exe` (but make sure the target, in this example "**final**", is the correct one) in `fbcli`. You can give the VDB capture as a command-line argument, or use the file browser that opens. Control the



resimulator using the Game Properties tab in the VDB.

Code export: select a body in the structure you're interested in (e.g. a body in a ragdoll) and click on the Export sub-system checkbox in the GameProperties tab; this copies to the clipboard a C++ function definition that creates the bodies connected to the selected body.