Dynamics Simulation

A whirlwind tour.

(Current State, and New Frontiers)

Russell Smith

Dec 2004

What I’ll Talk About

1. Dynamics simulation.
    - What is it?
    - Existing applications.
    - Technology (the whirlwind tour) .
2. Making simulation easier for the end user.
    - Technical challenges.
3. New techniques and applications.

Chapter 1

Dynamics Simulation

What is Dynamics?

- Classical physics: Newton’s law[s].

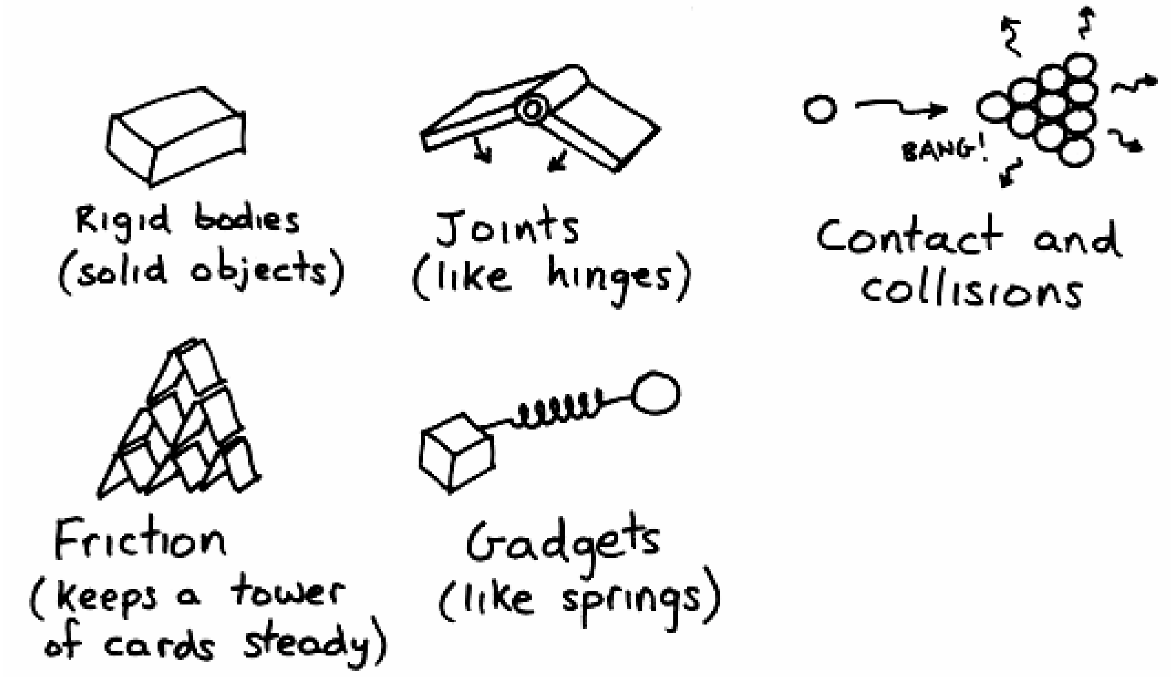
High school $\longrightarrow$ $f = ma$

$$\mathbf{M}\frac{\mathrm{d}^2\mathbf{p}}{\mathrm{dt}^2} = -\frac{\partial V}{\partial \mathbf{p}}^{\mathrm{T}}$$ $\longleftarrow$ Grad school

- In this talk “dynamics” is mostly “articulated rigid body dynamics”.
- But also:
  - Particle dynamics, cloth dynamics, wave dynamics, fluid dynamics, flexible body dynamics, fracture dynamics…

Dynamics Simulation Libraries

- API primitives: rigid or flexible bodies, joints, contact with friction, collision detection, etc…
- Many techniques, many libraries, lots of research.

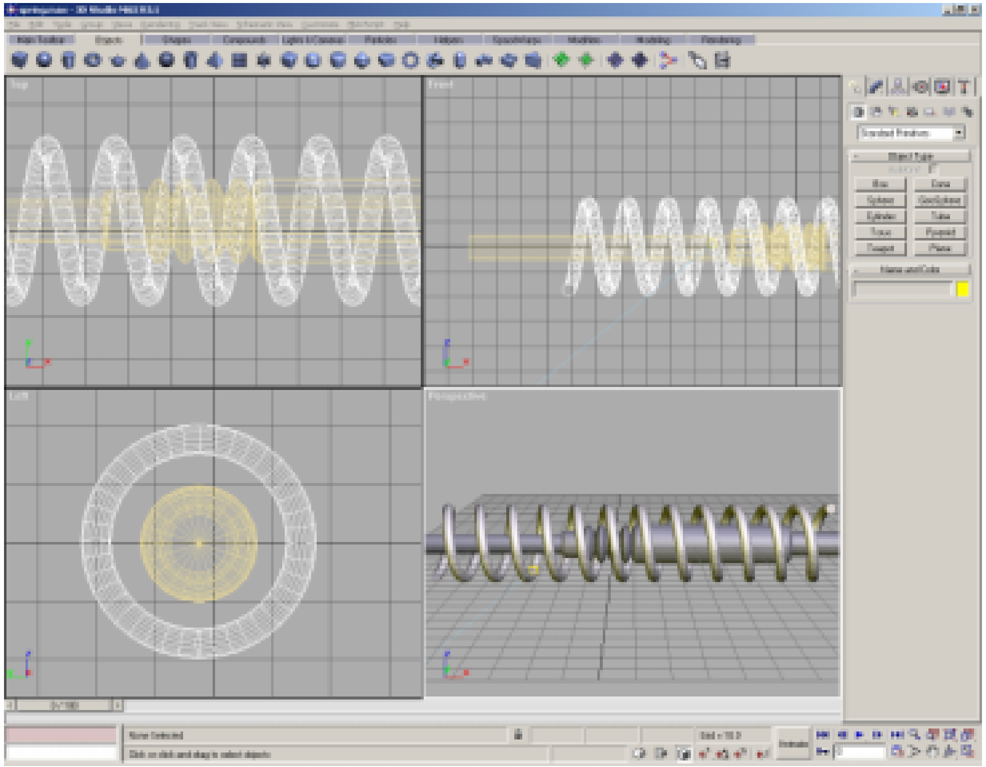


Applications: Games

- Interactive 3D worlds – typically FPS games.
- Limited uses: stacks of boxes, rag-dolls, collapsing buildings. But: need fast, stable and predictable simulation.

Applications: 3D animation tools

- The big three (Maya, SoftImage, 3DS Max), many others.
- Simulate dead things, or in combination with motion capture.
- Many custom tools, e.g. Massive (“Lord of the Rings”).



Applications: Industrial

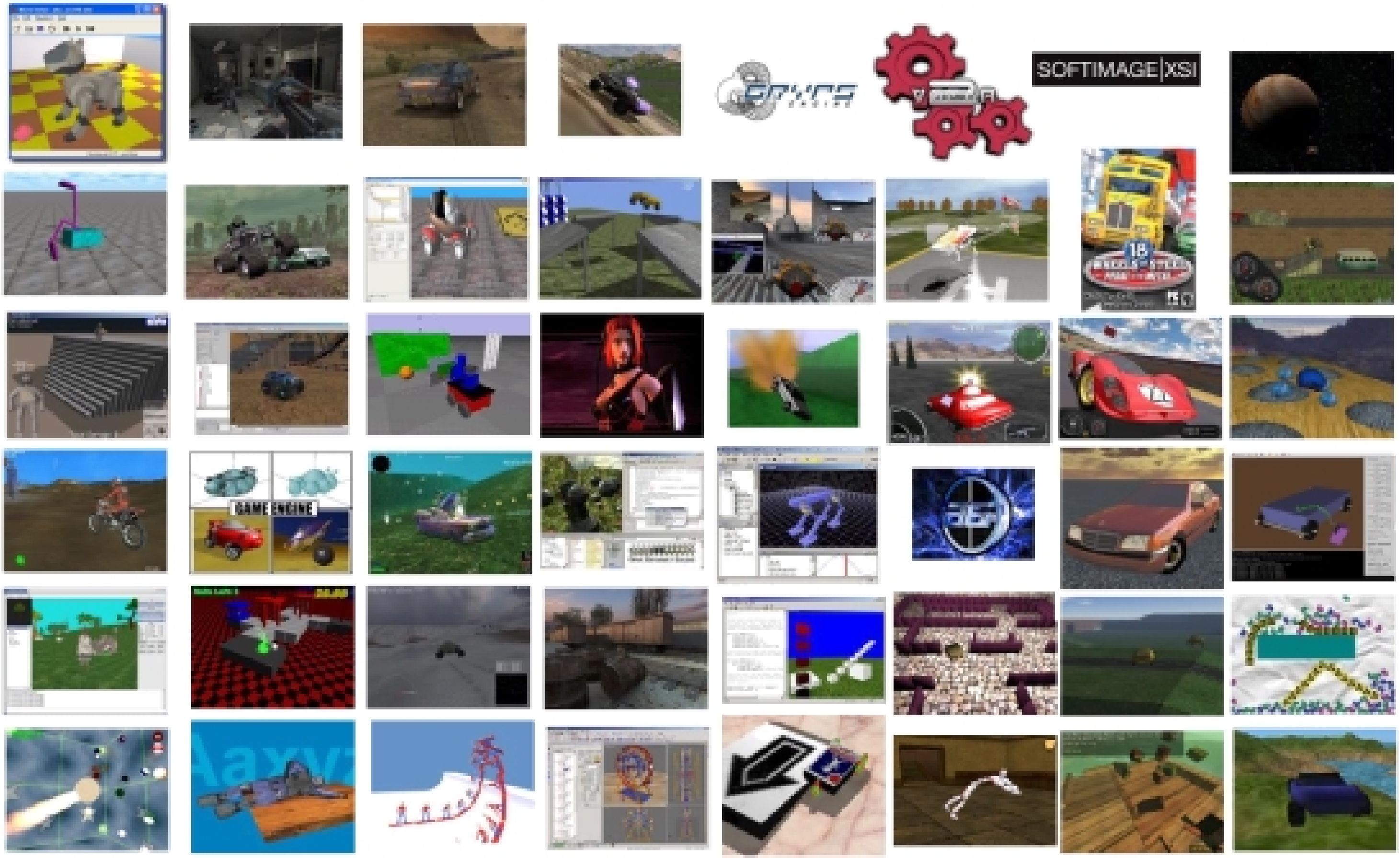
- Robot prototyping / modeling / research (e.g. Honda ASIMO, NASA mars rover).
- Biomechanics.
- Vehicle operator training, prototyping.

OPEN DYNAMICS ENGINE™

- My rigid body simulation library.
  - Many others: Havok, PhysX (Novodex, Meqon), SD/Fast…
- A platform for research.
  - Simulation algorithms, simulation applications.
- Open source (BSD license).
  - Dynamics should be ubiquitous: encourage innovation.
  - Closed source libraries constrain users: endless customization and integration hassles.
  - Why customization: ODE → SoftImage XSI → ILM
    - Used in “Eternal Sunshine of the Spotless Mind”.

OPEN DYNAMICS ENGINE™

- Over 1000 users: widely used in games, game engines, robot simulation, 3D animation.



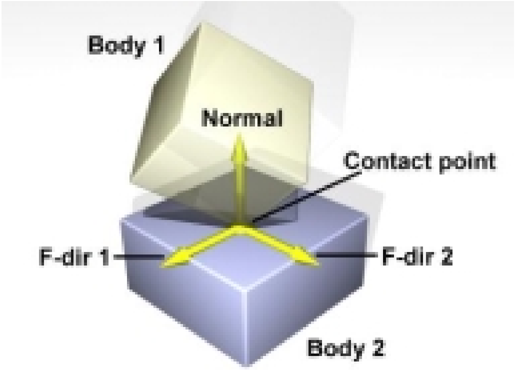
Technology #1 – Equations of Motion

- Lagrange multiplier (LM).
  - State space includes all body degrees of freedom. Model constraint forces explicitly. Invert a big matrix in each step.
- Reduced coordinate (RC).
  - State space has minimum size. Constraint forces are implicit. Spatial algebra. Fast tree-based factorization.
- State of the art: hybrids.
  - LM more flexible, RC faster – combine the two.
- Jakobsen – the easy way.
  - Particle system, Verlet integrator. Fast but limited.
- Impulse dynamics – also a popular choice.
- Transformations of constraint manifold.
- Many more…

Technology #2 – Integration

- Integrator type.
  - Explicit: acceleration a function of current forces.
    - Numerical energy gains.
  - Implicit: acceleration a function of future forces.
    - Numerical damping, therefore more stable.
- Accuracy and stability.
  - Higher order = more stable (usually).
  - Higher order = more accurate? (it depends…)
- Integrator or time-stepper
  - Integrator: plug in $dy/dt=f(y)$
  - Time stepper: integrator mixed with model.
- State of the art: second order time steppers with pyramidal friction [Stewart and Trinkle].
- Symplectic integration – but what about non-conservative systems?

Technology #3 – Contact and Friction



- Old way: spring and damper.
  - Spring and damper prevent penetration.
  - Tangential damping gives viscous friction.
- New way: constraint based.
  - Non-penetration constraints, solve LCP.
- Constraint based friction.
  - Coulomb friction cones ideal – but solution may not exist!
  - Static and dynamic friction.
  - Approximating friction is the big problem: friction pyramids, friction boxes, time-steppers.
- Impact modeling
  - Velocity constraints give impulses for free.
- Frontiers: differential inclusions, non-convex LCP.

Technology #4 –LCP

- Lagrange multiplier technique: Solve the linear complementarity problem (LCP):

$$\mathbf{A}\mathbf{x}=\mathbf{b}+\mathbf{w},\quad \mathbf{x}\geq\mathbf{0},\quad \mathbf{w}\geq\mathbf{0},\quad \mathbf{x}^{\mathrm{T}}\mathbf{w}=\mathbf{0}$$

- If x=0 this is just factorization.
- Otherwise it’s a discrete optimization problem.
  - Find the subset of variables in **x** to clamp to zero.
  - If **A** is SPD, a single unique solution exists, otherwise existence and uniqueness is not guaranteed.
  - Various search and factorization-based algorithms, both direct and iterative. Jury still out on the “best” technique for RBD.
- Frontiers: Interior point methods, multi-grid LCP, non-convex LCP, nonlinear-CP.

Technology #5 – Optimization

- For Lagrange multiplier technique, must factorize the “system matrix” (also principle sub-matrices).
- Matrix may be singular (PSD) or close to it.
  - Must use numerically robust algorithms.
- It’s not about MFLOPS, it’s about MBytes/s
  - Naïve implementations starve floating point units, as main memory bandwidth is very small compared to peak MFLOPS.
  - Many tricks: minimize memory traffic, cache-friendly algorithms, pipelining, SIMD, ATLAS.
- Frontiers:
  - Multi-frontal solvers for sparse matrices.
  - Reduced coordinate hybrid techniques.
  - Iterative solvers.
  - GPUs, custom hardware.

Technology #6 – Collision

- Collision detection an entirely separate field.
  - Computational geometry.
  - *Many* problems to solve.
- Standard approaches:
  - Primitive–to–primitive, $O(N^2)$ functions to write.
  - GJK / VClip (convex polyhedra).
  - Culling (AABBs, OBBs, BSPs, etc).
  - Triangle mesh (RAPID).
- More than just intersection tests.
  - Generate contact points, contact normals and depths.
  - Contact culling for good physical behavior.
  - Frontiers: Continuous collision detection to prevent interpenetration.

Chapter 2

Making Simulation Easier to Use

(and therefore cheaper)

Why Simulation is Hard

- Modeling real-world mechanisms is hard.
- Unexpected behavior.
  - Hard-to-debug numerical explosions, jitter, poor contact behavior and general unexpected weirdness.
- APIs force the user to learn arcane concepts.
  - Many simulation primitives not intuitive – angular velocity, inertia tensors.
- Too slow for big models.
- Force-based modeling is tricky.
- Too many numerical parameters to tune.
  - Many modeling / numerical approximations used, all with their own tradeoff parameters. Little guidance available, need to experiment.

Case Study: Game Worlds

- Don’t have to script all behavior (easier content creation?)
  - But: hard to script *any* behavior. Lack of artist control.
- Improved realism and world consistency.
  - Objects “do the right thing” – but only if the model is good, and this might not be what you want anyway (e.g., vehicle centers of mass).
- Emergent behavior.
  - Often pleasantly surprising, often annoying. Game design must allow for unpredictable outcomes.
- Players can exercise more creativity and control.
  - World building, story telling.
- Other problems:
  - Extra control / AI needed for virtual creatures.
  - Allow a dynamic player to participate in a dynamic environment.
  - More computation needed – eats the CPU budget.

Case Study: Robot Control

- Simulations are quicker to build than robots.
  - But *realistic* simulations are hard to build: need to model actuators (electrical motor dynamics, hydraulics / pneumatics, gear boxes, friction, stiction, flexion and slip), sensors, robot geometry and mass distribution, joint geometries, flexible bodies (vibration effects).
- Simulations let you cheat.
  - Easy to make controllers that exploit quirks of the simulated world, that don’t work so well in real life.
- Simulated robots don’t break.
  - The cost of experimentation *may* be lower.
- The best tradeoff:
  - Prototype robot control algorithms on a good-enough simulation, then move to the real hardware.

API Issues

- In the old days it was harder:
  - MDH parameters, weird reference frames, text file configuration, poor documentation, implementation exposed.
- Now we think about the user experience.
  - Absolute positions, utility functions (e.g. for rotation), interactive setup (3D tools), documentation, API consistency, only essential concepts in the API.
- Still lots of room for improvement.
  - Constructive modeling: glue, split, clone, deform, etc.
  - Dynamics debuggers – identify model physical / numerical errors.
  - Standardized data formats.

Speed

- Higher speed → real time simulation of more complex worlds.
- Big-matrix methods need lots of Optimization.
  - Coding tricks: minimize memory traffic, cache-friendly algorithms, pipelining, SIMD.
    - Parameterized code, search for efficient parameters (ATLAS).
- CPU budgeting.
  - Iterative methods allow us to cap effort per frame.
  - But accuracy is an issue.
- Parallelization.
  - Problem is not coarse grained – so clusters don't work well.
  - Parallel direct factorization – only for *large* problems.
  - Iterative techniques the easiest to parallelize.
  - ODE QuickStep inner loop: 3x speedup using 6 CPUs – [SGI Altix].

Force Based Modeling (bad!)

- User calculates and applies forces to bodies, e.g.:
  - Contacts and friction: spring and damper contact points.
  - Actuators and brakes: PD control of joint forces.
- Why it's bad:
  - Lots of parameters to tune (spend all your time searching a high-dimensional parameter space).
  - Usually hard to achieve desired effects (e.g. non-penetration of contacts).
  - Stiff forces react badly with explicit integrators.
    - Implicit integrators are slow on user-supplied forces.

Constraint Based Modeling (good!)

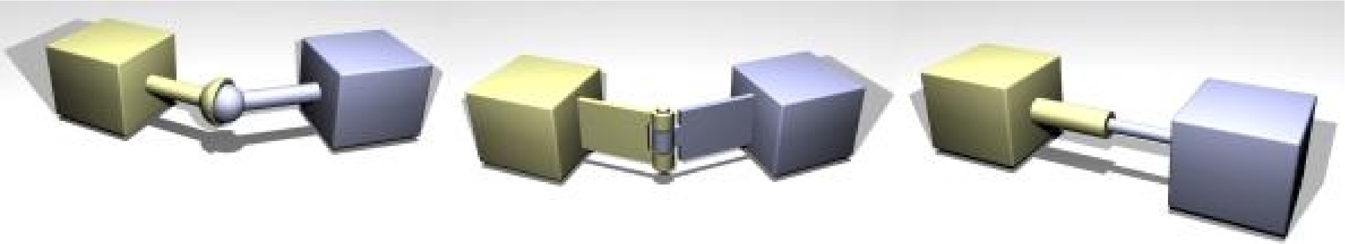
- Velocity / acceleration [in]equality constraints (LCP):

$$f_u(\mathbf{v}) = 0,\ f_v(\mathbf{v}) \ge 0 \quad \text{or} \quad f_a(\mathbf{a}) = 0,\ f_b(\mathbf{a}) \ge 0$$

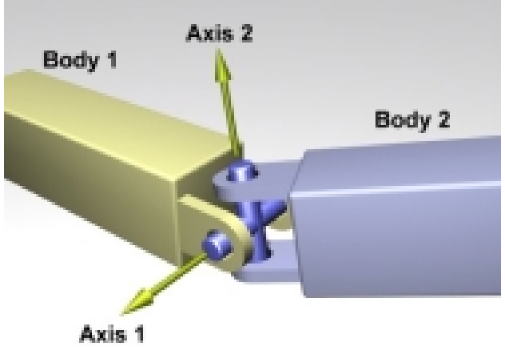
- Contacts and friction.
  - Relative velocity & force normal to contact surface ≥ 0.
  - Tangential forces limited by Coulomb friction (various models).
  - Constraint modeling is now commonplace for contacts.
- Actuators and brakes.
  - Joint velocity = v, but don’t apply too much force.
- Simulator enforces constraints automatically
  - No parameters to tune.
  - Integration problems hidden away.
- Can also model stiff springs, e.g. suspensions.

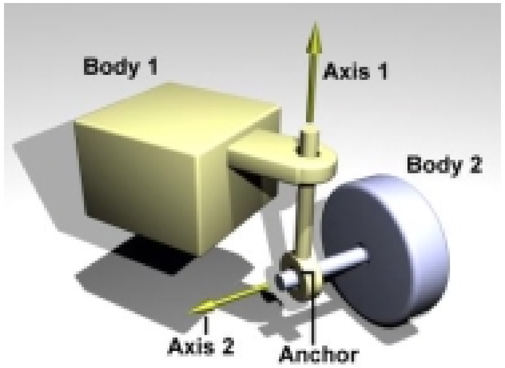
Joints are Constraint

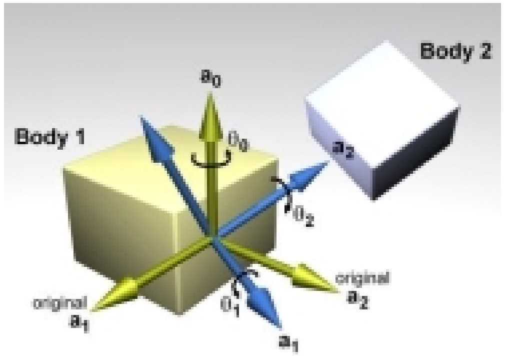
ODE’s “robot” joints:



ODE’s special purpose joints:

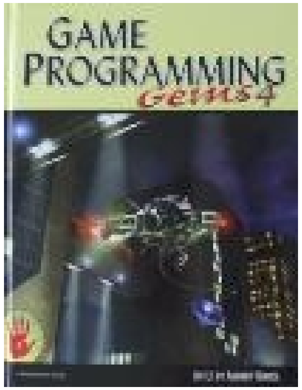






Constraint Also Good For:

- Mechanisms
  - – Gears, linked platforms, steering geometry, suspensions, roller coasters, weird joints (e.g. screw joints), etc.
- Modeling
  - – Contact geometry, various kinds of friction, various actuators, spongy / flexible joints, etc.
- Disadvantages:
  - – More expensive than forces.
    - Must factor a matrix of constraint information.
  - – More mathematically difficult to formulate.
    - But “intuitive” guidance available, see Game Gems IV ch3.4.



Chapter 3

New Techniques and Applications

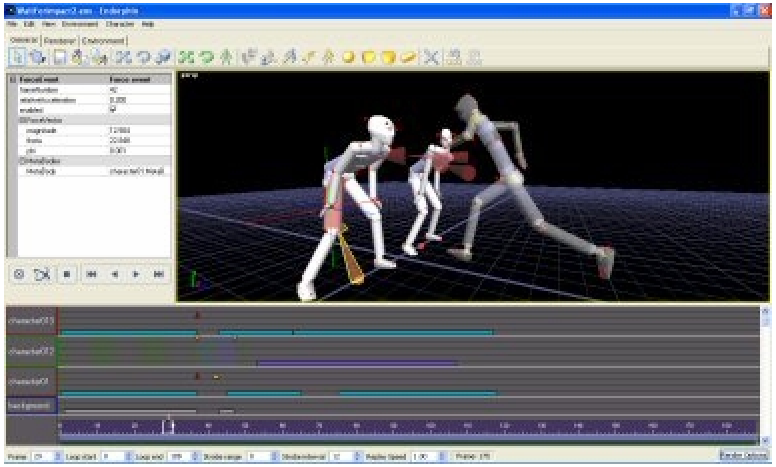
New Ways of Using Simulation

- Animation Tools.
  - Artist control.
  - Ghosting
  - Motion retargeting.
  - Motion scripting, time-extended constraints.
  - Realistic movement: alternatives to key-framing / MoCap.

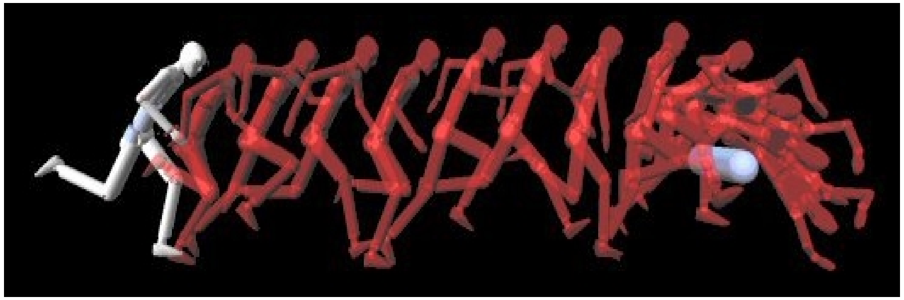
- Robot Control.
  - Simulation-in-the-loop
  - Behavioral constraints, e.g. balancing.
  - Motion retargeting.
  - Model fitting (converge on robot model).
  - Look-ahead, look-ahead constraints.
- Other categories:
  - Haptics (e.g. virtual surgery), biomechanics research / diagnosis, industrial prototyping (vehicles, robots), product presentation, operator training.

Artist Control #1: Kinematic Control

- First order dynamics used for posing (click+drag positioning).
  - First order: velocity (not acceleration) driven by force.
  - Easier than inverse kinematics, all constraints in the toolbox can be applied.
  - E.g.: Endorphin (www.naturalmotion.com)



Also: ghosting (Endorphin, XSI)

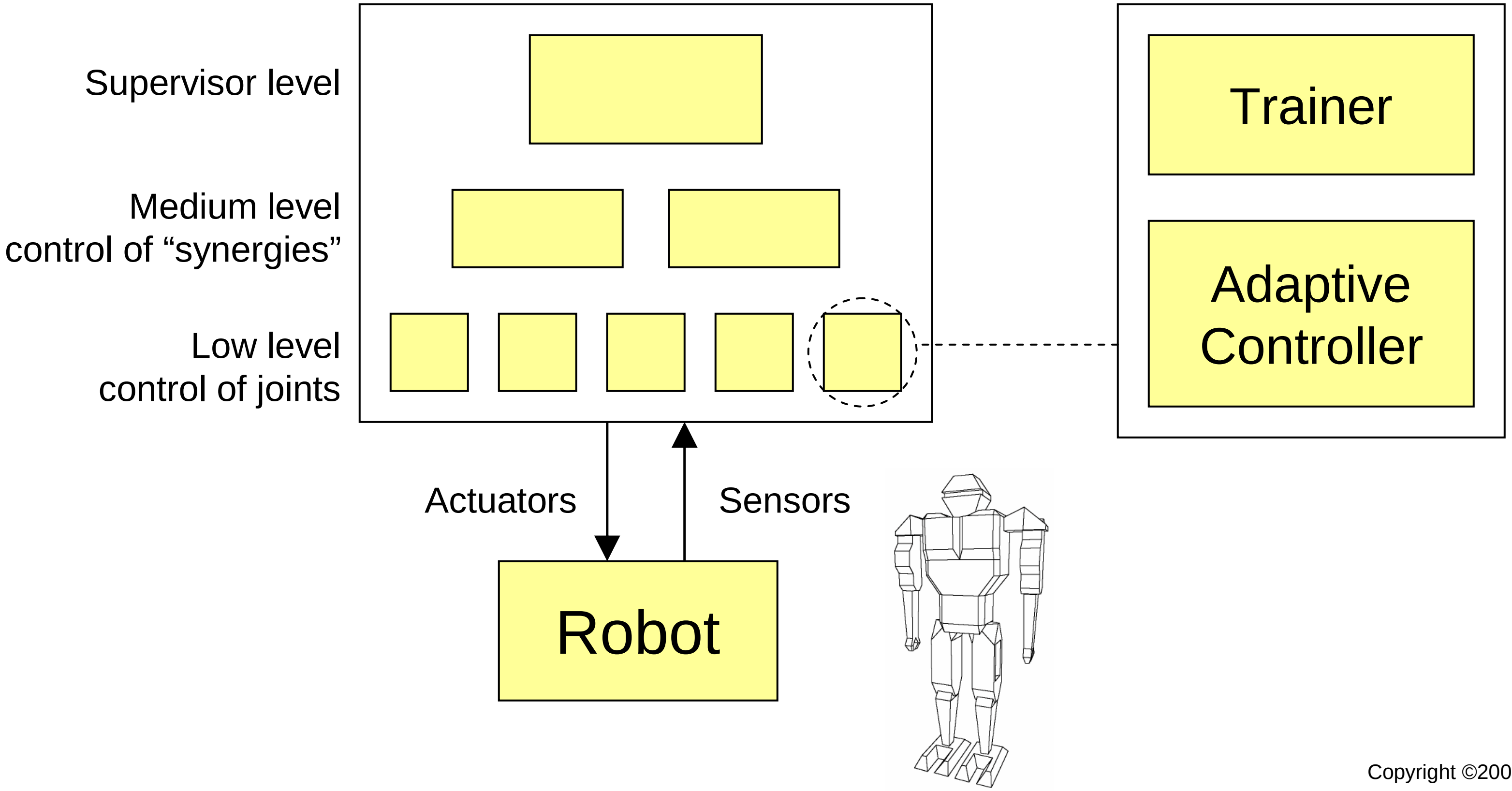


Artist Control #2: Motion Control

- Motion retargeting.
  - Source motion drives end effector constraints, or inverse collision constraints.
  - Constraints added to target structure so that it follows the *relevant* features of the source motion.
- Motion scripting.
  - Constrain the end-state of a simulation, or a functional of the trajectory. Get high level motion control.
  - Point-and-shoot techniques. Dynamic programming?
  - Hybrid systems - combine kinematics (MoCap) with dynamics.
  - Need good designer interfaces - still experimental.

Simulation-in-the-Loop (to Augment Intelligent Control)

- A popular approach: modular, hierarchical design.
  - “Biologically inspired”.
  - Adaptive control used in lower levels.
  - E.g.:



Supervisor level

Medium level control of “synergies”

Low level control of joints

Trainer

Adaptive Controller

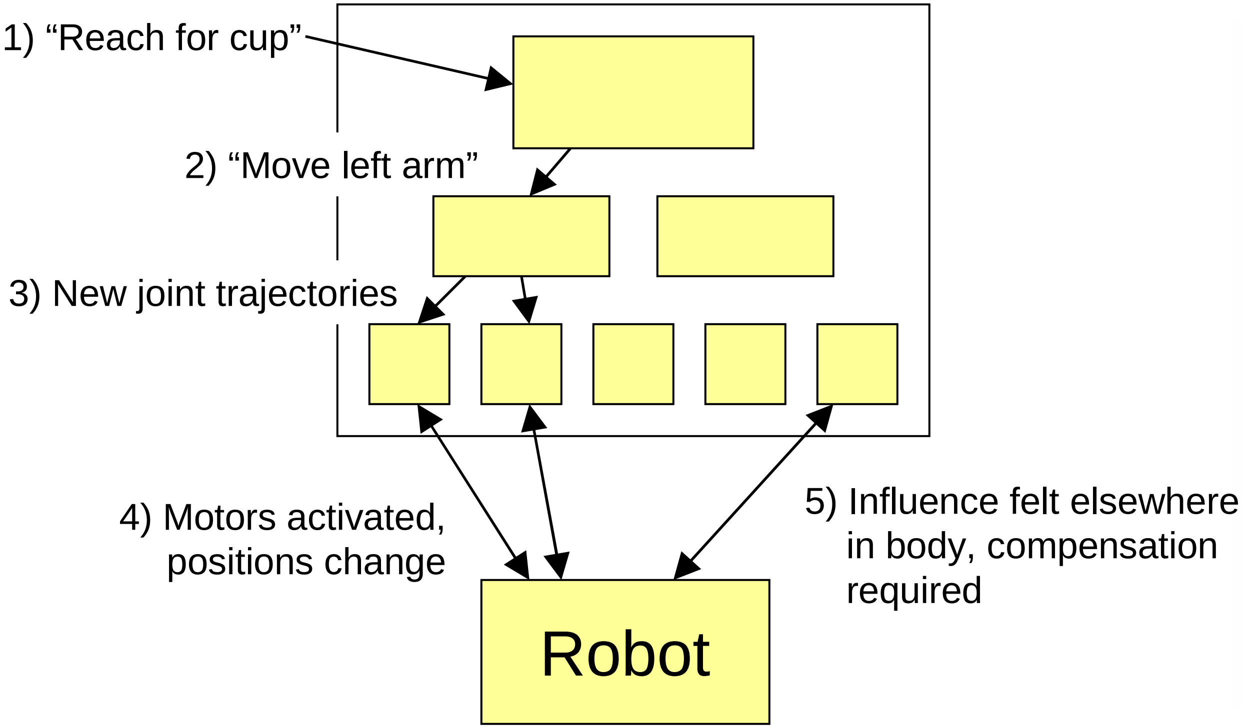
Actuators

Sensors

Robot

A Common Problem: Compensation For Internal Disturbance.

- A high level command descending through the hierarchy may inadvertently affect other parts of the hierarchy, e.g.:



1) “Reach for cup”

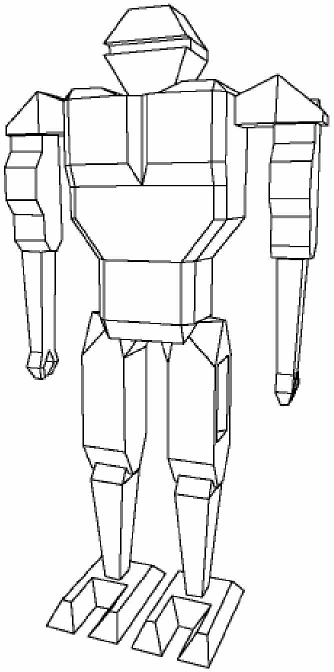
2) “Move left arm”

3) New joint trajectories

4) Motors activated, positions change

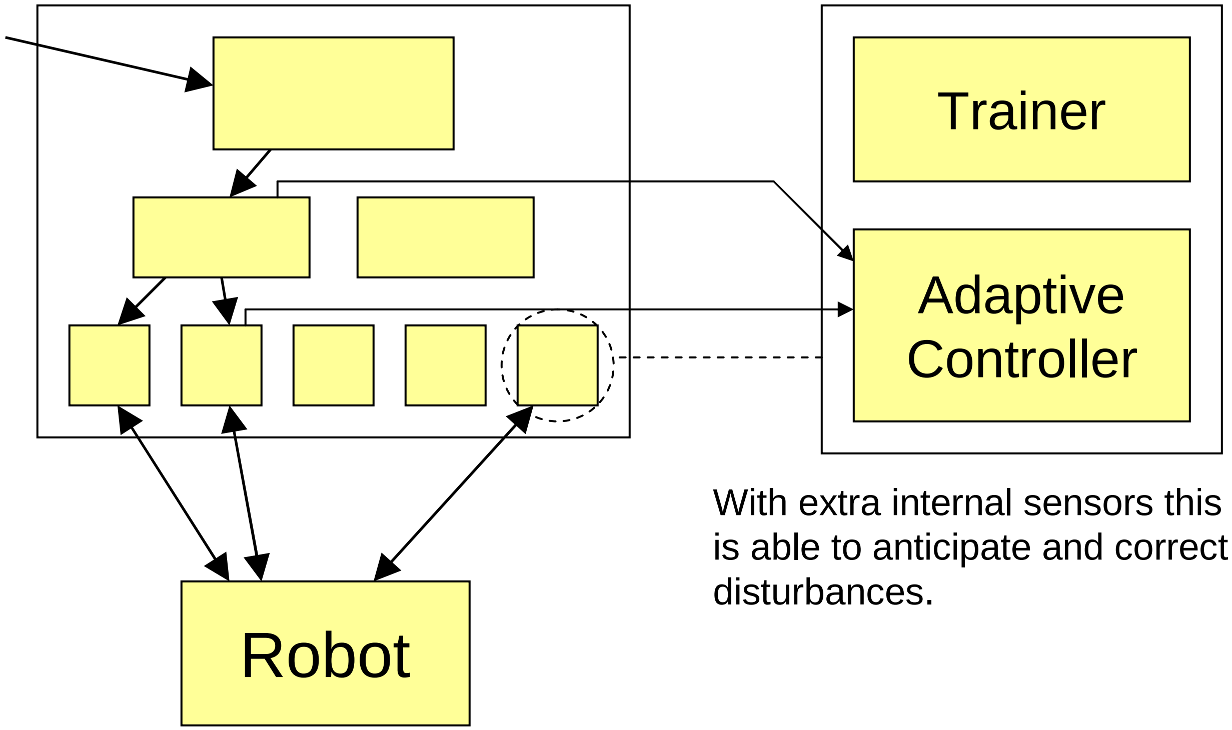
5) Influence felt elsewhere in body, compensation required

Robot



Standard Solutions

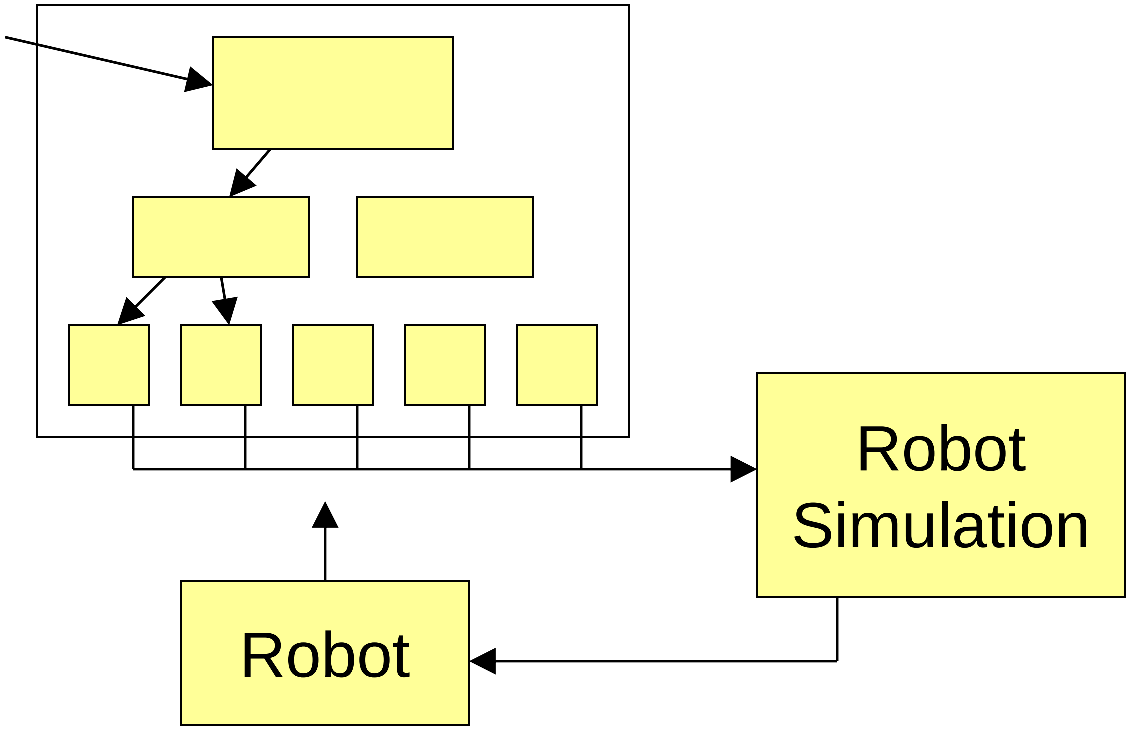
1. Add extra associativity to the low level controllers.
2. Use global inverse dynamics to determine joint forces.
    - Lack of flexibility, e.g. fully specified motion for all joints – a problem if we want joint compliance.



With extra internal sensors this unit is able to anticipate and correct for disturbances.

Simulation-in-the-Loop

- Simulation input: joint force / velocity reference.
- Joint forces read out.
- Constraints automatically compensate for disturbances.



Behavioral Constraints

- E.g. balancing: control center of mass projection onto floor:

$$\frac{\sum_i m_i \, \mathbf{n}^{\mathrm{T}} p_i}{\sum_i m_i} = \text{whatever} \quad (\mathbf{n} \text{ is floor normal})$$

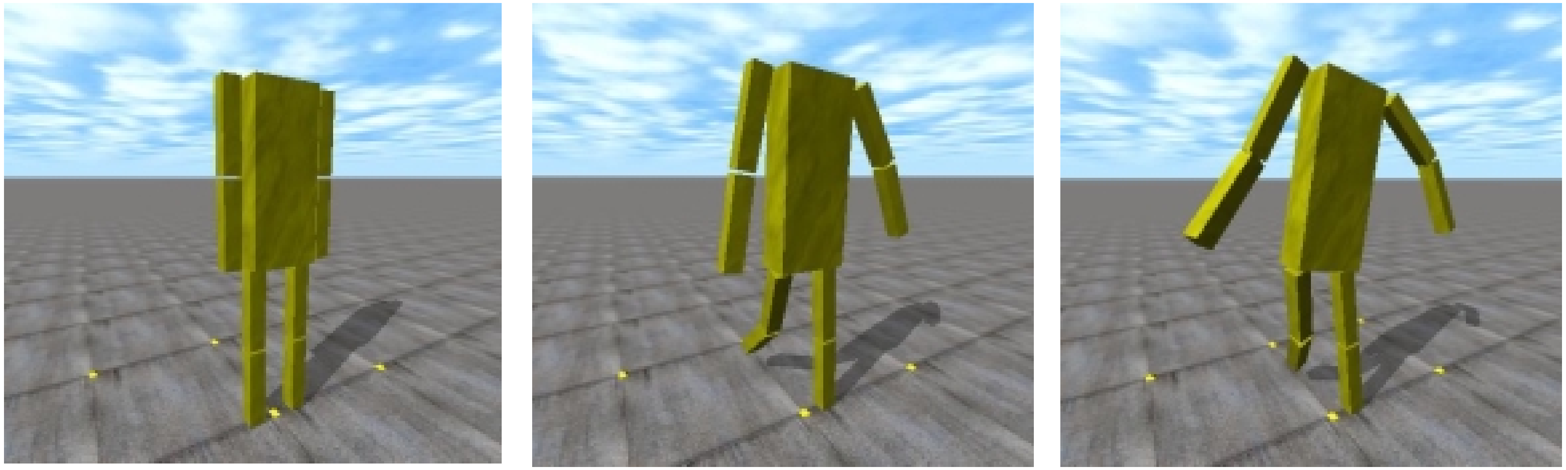
- We can express this as a velocity constraint:

$$\text{momentum} = \mathbf{n}^{\mathrm{T}} \sum_i m_i \, v_i = \left(\sum_i m_i \right) \frac{\mathrm{d}}{\mathrm{dt}} \text{whatever}$$

- The trick: compute the actuator forces that could substitute for the constraint forces..
    - Least squares problem.
    - Can not always find a solution, e.g. depends on initial conditions.

Example: Balancing Constraint

- Simple biped model:
  - Left foot anchored to ground.
  - The only control rule: keep left leg straight.
  - Right leg dragged by an external force.
  - Balance constraint implies actuator commands → biped posture changes to keep balance.



The End