Game Physics

Game and Media Technology
Master Program - Utrecht University

Dr. Nicolas Pronost

Physics engine
design and implementation

Physics engine

- The physics engine is a component of the game engine
- The game engine separates reusable features and specific game logic
  - basically software components (physics, graphics, input, network, *etc*.)
- The physics engine handles the simulation of the world
  - physical behavior, collisions, terrain changes, ragdoll and active characters, explosions, object breaking and destruction, liquids and soft bodies, …

Physics engine

- Some SDKs
  - Open Source
    - Bullet, Open Dynamics Engine (ODE), Tokamak, Newton Game Dynamics, PhysBam, Box2D
  - Closed source
    - Havok Physics
    - Nvidia PhysX

PhysX (Mafia II)

ODE (Call of Juarez)

Havok (Diablo 3)

Case study: Bullet

- Bullet Physics Library is an open source game physics engine
  - http://bulletphysics.org, open source under ZLib license
  - It provides collision detection, soft body and rigid body solvers
  - It has been used by many movie and game companies in AAA titles on PC, consoles and mobile devices
  - It has a modular extendible C++ design
  - This is the engine you will use for the practical assignment
    - have a good look at the user manual and the numerous demos (*e.g.* CCD Physics, Collision and SoftBody Demo)

Features

- Bullet Collision Detection can be used on its own as a separate SDK without Bullet Dynamics
  - Discrete and continuous collision detection
  - Swept collision queries
  - Generic convex support (using GJK), capsule, cylinder, cone, sphere, box and non-convex triangle meshes
  - Support for dynamic deformation of non-convex triangle meshes
- Multi-physics Library includes
  - Rigid body dynamics including constraint solvers
  - Support for constraint limits and motors
  - Soft body support including cloth and rope

Design

- The main components are organized as follows

Soft Body Dynamics

Bullet Multi Threaded

Extras:
Maya Plugin,
etc.

Rigid Body Dynamics

Collision Detection

Linear Math, Memory, Containers

Overview

- First the high level simulation manager is defined
  - `btDiscreteDynamicsWorld` or `btSoftRigidDynamicsWorld`
  - manages the physics objects and constraints
  - implements the update call to all objects at each frame
- Then the objects are created
  - `btRigidBody`
  - you will need
    - the mass (>0 for dynamic objects, 0 for static)
    - the collision shape (box, sphere, *etc.*)
    - the material properties (friction, restitution, *etc.*)
- Finally the simulation is updated at each frame
  - `stepSimulation`

Initialization

```
// Collision configuration contains default setup for memory, collision setup
btDefaultCollisionConfiguration * collisionConfiguration = new
   btDefaultCollisionConfiguration();

// Set up the collision dispatcher
btCollisionDispatcher * dispatcher = new
   btCollisionDispatcher(collisionConfiguration);

// Set up broad phase method
btBroadphaseInterface * overlappingPairCache = new btDbvtBroadphase();

// Set up the constraint solver
btSequentialImpulseConstraintSolver * solver = new
   btSequentialImpulseConstraintSolver();

btDiscreteDynamicsWorld * dynamicsWorld = new btDiscreteDynamicsWorld(dispatcher
   , overlappingPairCache, solver, collisionConfiguration);

dynamicsWorld->setGravity(btVector3(0,-9.81,0));
```

Simulation

```
for (int i=0; i<100; i++) {
    dynamicsWorld->stepSimulation(1.0f/60.f, 10);

    // print positions of all objects
    for (int j=dynamicsWorld->getNumCollisionObjects()-1; j>=0 ; j--) {
        btCollisionObject * obj = dynamicsWorld->getCollisionObjectArray()[j];
        btRigidBody * body = btRigidBody::upcast(obj);
        if (body && body->getMotionState()) {
            btTransform trans;
            body->getMotionState()->getWorldTransform(trans);
            printf("World pos = %f,%f,%f\n",
    float(trans.getOrigin().getX()), float(trans.getOrigin().getY()),
    float(trans.getOrigin().getZ()));
        }
    }
}
```

Termination

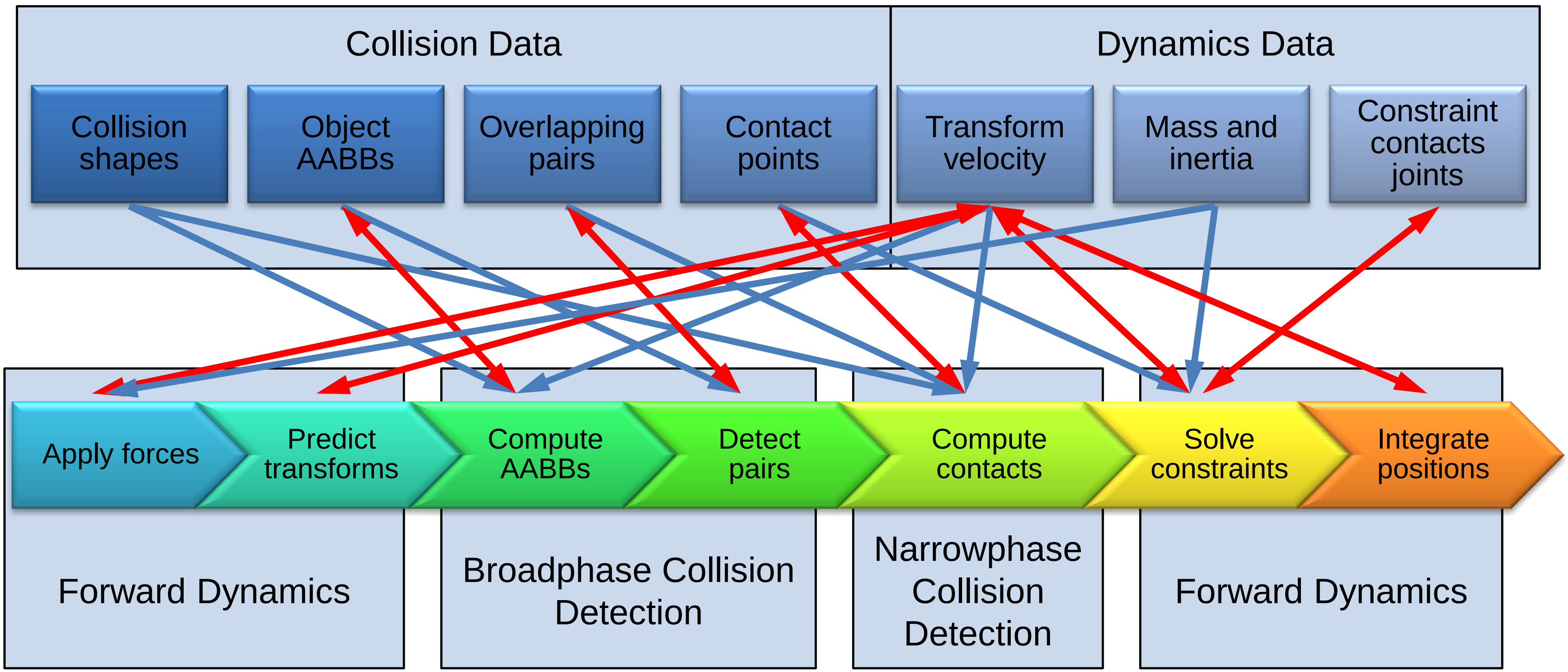
```
//remove the rigid bodies from the dynamics world and delete them
for (int i=dynamicsWorld->getNumCollisionObjects()-1; i>=0 ; i--) {
    btCollisionObject * obj = dynamicsWorld->getCollisionObjectArray()[i];
    btRigidBody * body = btRigidBody::upcast(obj);
    if (body && body->getMotionState()) delete body->getMotionState();
    dynamicsWorld->removeCollisionObject(obj);
    delete obj;
}

// delete collision shapes
for (int j=0; j<collisionShapes.size(); j++) {
    btCollisionShape * shape = collisionShapes[j];
    collisionShapes[j] = 0;
    delete shape ;
}

delete dynamicsWorld;
delete solver;
delete overlappingPairCache;
delete dispatcher;
delete collisionConfiguration;
```

Rigid Body Physics Pipeline

- Data structures used and computation stages performed by a call to `stepSimulation`



Simulation step

- The simulation stepper updates the world transformation for active objects by calling `btMotionState::setWorldTransform`
- It uses an internal fixed time step of 60 Hertz
  - when the game frame frequency is smaller (game faster), it interpolates the world transformation of the objects without performing simulation
  - when the game frame frequency is larger (game slower), it will perform multiple simulations
    - the maximum number of iterations can be specified

Collision detection

- Bullet provides algorithms and structures for collision detection
  - Object with world transformation and collision shape
    - `btCollisionObject`
  - Collision shape (box, sphere *etc*.) usually centered around the origin of their local coordinate frame
    - `btCollisionShape`
  - Interface for queries
    - `btCollisionWorld`
- The broad phase quickly rejects pairs of objects that do not collide using a dynamic bounding volume tree based on the AABBs
  - it can be changed to another algorithm

Collision dispatcher

- A collision dispatcher iterates over each pair of possibly colliding objects, and calls the collision algorithm corresponding to each configuration
- These algorithms return the time of impact, the closest points on each object and the penetration depth / distance vector

Collision dispatcher

	BOX	SPHERE	CONVEX, CYLINDER, CONE, CAPSULE	COMPOUND	TRIANGLE MESH
BOX	boxbox	spherebox	gjk	compound	concaveconvex
SPHERE	spherebox	spheresphere	gjk	compound	concaveconvex
CONVEX, CYLINDER, CONE, CAPSULE	gjk	gjk	gjk	compound	concaveconvex
COMPOUND	compound	compound	compound	compound	compound
TRIANGLE MESH	concaveconvex	concaveconvex	concaveconvex	compound	gimpact

Collision detection

- Bullet uses a small collision margin for collision shapes to improve performance and reliability
  - set to a factor of 0.04 (i.e. expand the shape by 4 cm if unit is meter)
  - to still look correct, the margin is usually subtracted from the original shape

- It is always highly recommended to use SI units everywhere

User collision filtering

- Bullet provides three ways to filter colliding objects
  - Masks
    - user defined IDs (could be seen as layers in 2D) grouping possibly colliding objects together
  - Broadphase filter callbacks
    - user defined callbacks called at the early broad phase of the collision detection pipeline
  - Nearcallbacks
    - user defined callbacks called at the late narrow phase of the collision detection pipeline

Rigid body dynamics

- The rigid body dynamics is implemented on top of the collision detection
- It adds force, mass, inertia, velocity and constraint
- Main rigid body object is `btRigidBody`
  - moving objects have non-zero mass and inertia
  - inherits world transform, friction and restitution from `btCollisionObject`
  - adds linear and angular velocity

Rigid body dynamics

- Bullet has 3 types of rigid bodies
  - Dynamic (moving) bodies
    - have positive mass, position updated at each frame
  - Static (non moving) bodies
    - have zero mass, cannot move but can collide
  - Kinematic bodies
    - have zero mass, can be animated by the user (can push dynamic bodies but cannot react to them)

Rigid body dynamics

- The world transform of a body is given for its center of mass
  - if the collision shape is not aligned with COM, it can be shifted in a compound shape
- Its basis defines the local frame for inertia
- The `btCollisionShape` class provides a method to automatically calculate the local inertia according to the shape and the mass
  - the inertia can be edited if the collision shape is different from the inertia shape

Rigid body dynamics

- Rigid body constraints are defined as `btTypedConstraint`
  - Bullet includes different constraints such as hinge joint (1 rot. DOF) and ball-and-socket joint (3 rot. DOF)
- Constraint limits are given for each DOF
  - Lower limit and upper limit
  - 3 configurations
    - lower = upper means that the DOF is locked
    - lower > upper means that the DOF is unlimited
    - lower < upper means that the DOF is limited in that range

Soft body dynamics

- Bullet provides dynamics for rope, cloth and soft body
- The main soft body object is `btSoftBody` that also inherits from `btCollisionObject`
  - each node has a dedicated world transform
- The container for soft bodies, rigid bodies and collision objects is `btSoftRigidDynamicsWorld`

Soft body dynamics

- Bullet offers the function `btSoftBodyHelpers::CreateFromTriMesh` to automatically create a soft body from a triangle mesh
- Bullet can use either direct nodes/triangles collision detection or a more efficient decomposition into convex deformable clusters

Soft body dynamics

- Forces can be applied either on every node of a body or on an individual node

```
softBody->addForce(const btVector3& forceVector);
softBody->addForce(const btVector3& forceVector, int node);
```

- It is possible to make nodes immovable

```
softBody->setMass(int node,0.0f);
```

- Or attach nodes to a rigid body

```
softBody->appendAnchor(int node, btRigidBody* rigidbody, bool
    disableCollisionBetweenLinkedBodies=false);
```

- Or attach two soft bodies using constraints

Demos

- Convex collision

- Concave collision

- Convex hull distance

- Joint

- Fracture

- Soft

Assignment

- You will use Bullet in your assignment to control the motion of a creature
- The default configuration of the physics world uses
  - A 3D axis sweep and prune broad phase
  - A sequential impulse constraint solver
  - A fixed collision object for the ground
- The `Application` creates and manages a `Creature`, a `Scene` and the simulation time stepping
- The `Application` takes care of the simulation loop (update and render) and manages the user inputs
- The `Scene` manages the rotation of the mobile platform and the throwing of the balls

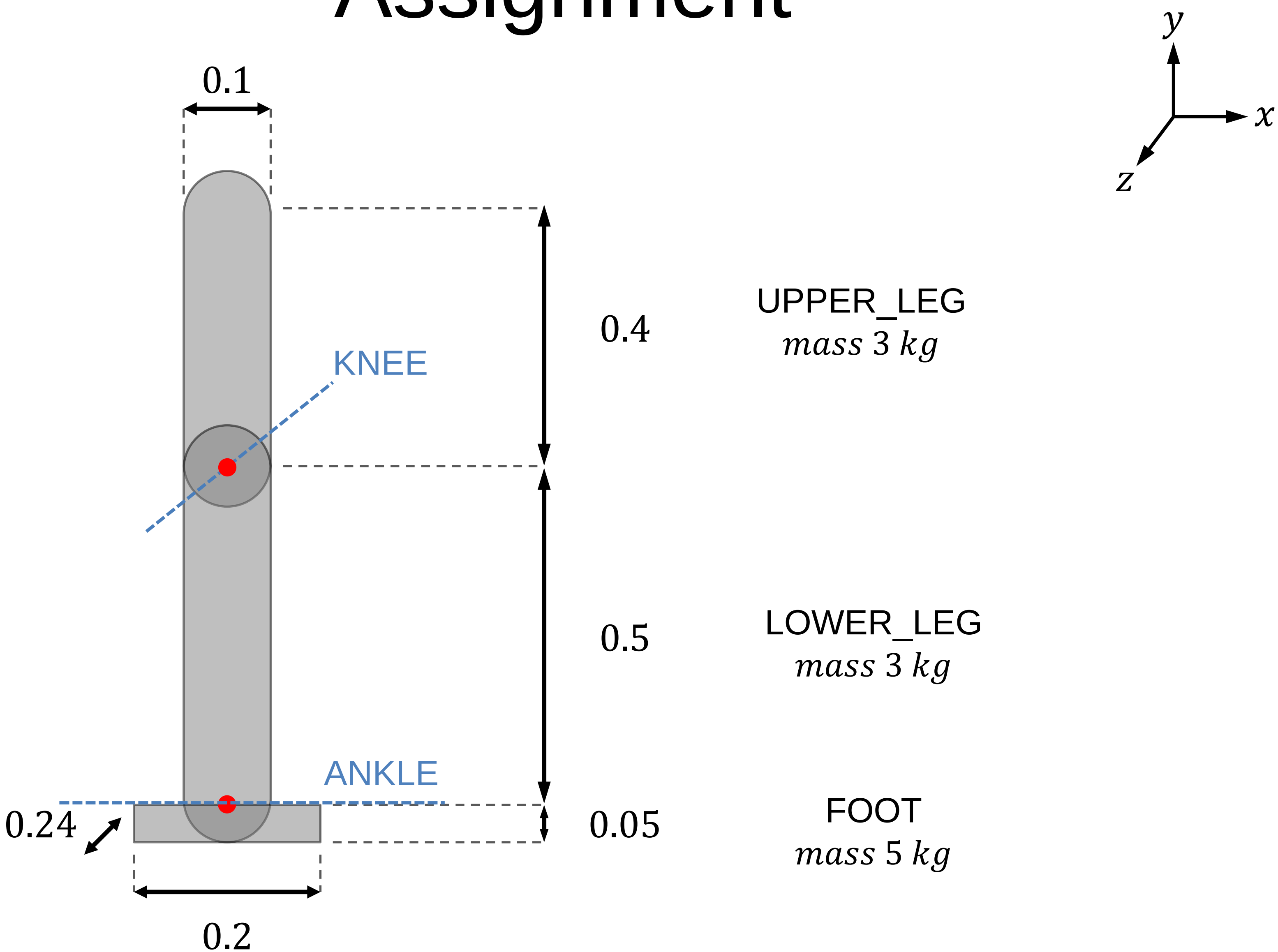
Assignment

- To control the motion of the creature you have to use PD controllers at the joints
  - Create a class `PDController` and add a container for them in the `Creature` (1 per DOF)
  - Angular motors have to be enabled for the joints you want to control (`Creature.cpp`, line 69 and 82)
  - PD controller gains have to be tuned to produce natural behavior
  - At each simulation step
    - The balance corrections are fed to the PD controllers
    - The PD controllers give back the torques to apply to correct the pose according to the current pose, velocity and gains
    - The torques are given to the joint motors (function `setMotorTarget`)

Assignment

- The function `btCollisionObject::getWorldTransform` returns a `btTransform` describing the 3D transformation from the local reference frame of an object to the global world reference frame (common to every object)
- The function `btTransform::inverse` can be used to get the inverse transformation
- The functions `getCenterOfMassPosition` and `getInvMass` return respectively the COM and the inverse of the mass of a `btRigidBody`

Assignment



0.1
KNEE
ANKLE
0.24
0.2
0.4
0.5
0.05
UPPER_LEG
$mass\ 3\ kg$
LOWER_LEG
$mass\ 3\ kg$
FOOT
$mass\ 5\ kg$
y
x
z

Efficiency

- Do not waste time with more processing power than needed to get a targeted effect
  - Graphics, AI, and so on need it as well
- Simplify the equations depending on the number of dimensions of the simulated world
- Use primitive shapes as much as possible for collision detection
  - use low number of vertices in convex hulls (performance and stability)

Efficiency

- Be careful about the ratios
  - sometimes difficult to manage both very small and very big objects, need to reduce internal time step
  - same for very different masses
- Combine multiple static triangle meshes into one to reduce computations in broad phase

Efficiency

- Neglect unwanted or not important effects
  - you can assume for example that the sum of the gravity, the reaction force and the static friction is zero
  - you can neglect or simulate air resistance by a drag coefficient multiplied by the velocity
- Run full physics simulation only on relevant objects
  - only visible or near player objects
  - only currently active objects
  - but be careful about the discontinuities when they are simulated again

Object (de)activation

- To save up many useless calculations, we do not want to simulate an object which does not move
  - For example sitting on the ground or a spring at rest
  - Because of drag and friction, only objects on which a consistent net force is applied will not settle down
- We need to come up with two functionalities
  - One for deactivating an object
  - And one for activating an object back

Object (de)activation

- Collision detector still returns contacts with deactivated objects but omitted in velocity resolution algorithm
  - Numerical integration is skipped for deactivated objects, so it saves computation time
- The object is deactivated when both linear and angular velocities are below a threshold (body specific values)
  - Deactivated objects are therefore more stable

Object (de)activation

- The object is activated
  - when it collides with another active object
    - another threshold can be used for the minimal severity of the collision needed to activate again the object
  - when non-constant external forces are applied to the object
- In a game, every object is initialized in its rest configuration and deactivated
  - At start up, it is then very fast, even with many objects
  - It is only when interactions occur with the object that it will be simulated until it settles down again

Optimization techniques

- Precompute as much as possible
  - Try to tabulate mathematical functions, random numbering *etc.*
  - To perform only array access in the physics update
  - Example
    - sine call takes 5 times longer to be evaluated than to access an array

```
float acc = 0;
for (int i = 0; i < 1000; i++)
   acc = acc + i * sin(x * i); // instead use: sinTable[x*i]
```

Optimization techniques

- Simplify your math
  - Mathematical operators are not equally fast
  - Complex function >> divide >> multiply >> addition/subtraction
  - Try to simplify equations (and/or tabulate them)
  - Try to reduce type conversion
  - Examples

```
double acc = 1000000;
for (int i = 0; i < 10000; i++) acc = acc / 2.0;
acc = 1000000;
for (int i = 0; i < 10000; i++) acc = acc * 0.5; // takes 60% of
   the execution time of the previous version
```

```
a*b + a*c = a*(b+c);      // gets rid of one multiply
b/a + c/a = (1/a)*(b+c); // changes one divide for one multiply
          = (b+c)/a;      // gets rid of one divide
```

Optimization techniques

- Store data efficiently
  - chose the right data type with the right precision
  - both code execution and memory footprint are proportional to the number of bytes used

Type	Size (B)	Range
char	1	[-128 , 127]
unsigned char	1	[0 , 255]
int	4	[-2 147 483 648 , 2 147 483 647]
unsigned int	4	[0 , 4 294 967 295]
float	4	[$-3.4*10^{38}$, $3.4*10^{38}$] (7 decimal)
double	8	[$-1.7*10^{308}$, $1.7*10^{308}$] (15 decimal)
bool	1	true / false

Optimization techniques

- Be linear
  - CPUs come with memory caches loaded when accessing data
  - Access continuous data in memory (*e.g.* traversing an array from begin to end) produces less cache misses
    - so less loading time
    - vectors are faster to traverse than lists

Optimization techniques

- Size does matter
  - To compile arrays of structures, the compiler performs a multiplication by the size to create the array indexing
    - if the structure size is a power of 2, the multiplication is replaced by a shift operation (much faster)
    - you can round array sizes aligned to a power of 2 even if you do not use all of it
  - Example

```
int softBodyNodes [38];
int softBodyNodes [64]; // faster allocation
```

End of
Physics engine
design and implementation

Next

Written exam