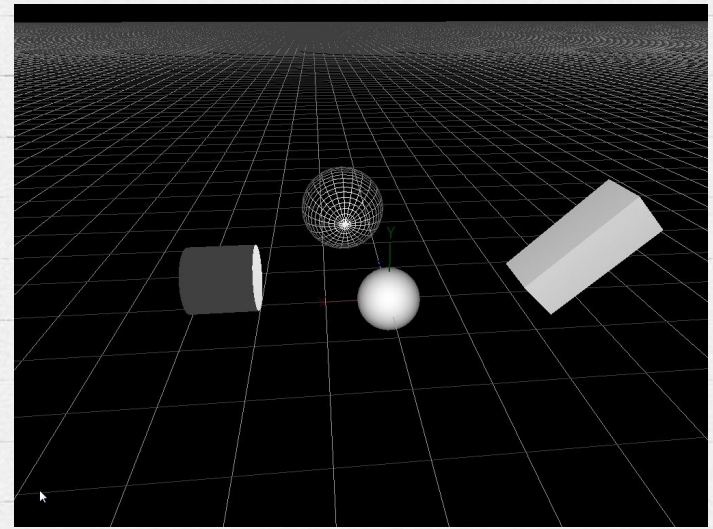

Bullet
integration

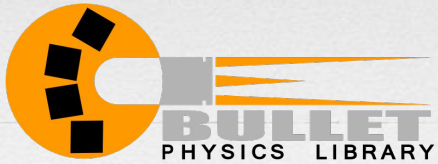
But first!

Should we review last week's assigment?

```
// TODO (Homework): Rotate the camera
with the mouse
```



Bullet



Bullet is a physics library created by Erwin Coumans. It supports collision detection and soft and rigid body dynamics.

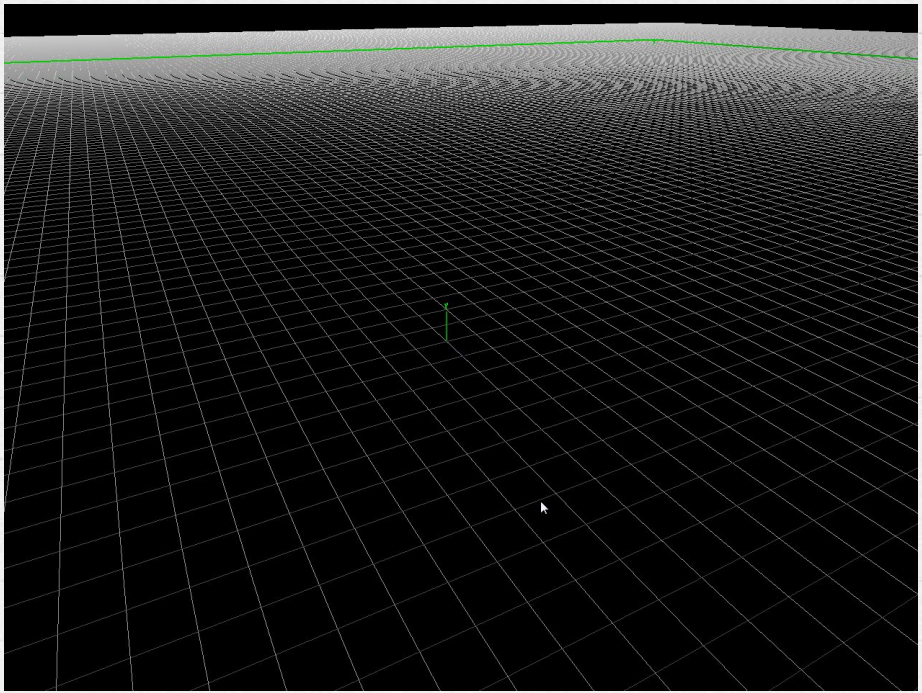
It's used in movies, videogames and other authoring tools.

In its github's repository you can find the manuals inside *docs* folder.

What you will have

By the end of the class

Hopefully



Let's do this!

TODO 1

```
// TODO 1: Add Bullet common include btBulletDynamicsCommon.h
// ...and the 3 libraries based on how we compile (Debug or Release)
// use the _DEBUG preprocessor define
```

- ✗ Add the 3 library files using #pragma comment(lib, "...")
  - ✗ Add the 3 "debug" and "release" versions
- ✗ Include "btBulletDynamicsCommon.h"

```
#if _DEBUG
	#pragma comment (...)
#else
	#pragma comment (...)
#endif
```

TODO 2

```
// TODO 2: Create collision_configuration, dispatcher,
// broad_phase and solver
// And destroy them!
```

A “bullet world” can be built with this classes, like “lego pieces”.

We’ll need all of them before we can create the world. But we can use the default ones provided by bullet already.

```
btDiscreteDynamicsWorld(btDispatcher* dispatcher, btBroadphaseInterface* pairCache,
btConstraintSolver* constraintSolver, btCollisionConfiguration* collisionConfiguration);
```

TODO 2

```
btDiscreteDynamicsWorld(btDispatcher* dispatcher, btBroadphaseInterface* pairCache,
btConstraintSolver* constraintSolver, btCollisionConfiguration* collisionConfiguration);
```

A **collision dispatcher** detects fine collisions and finds contact points. It also locates the adequate collision algorithm for each pair of objects.

Use **btCollisionDispatcher**

The **broadphase collision detection** does a first pass to detect object that "may collide". Simplifies all objects into spheres or boxes.

There's many ways to do this, we'll use **btDbvtBroadphase**

TODO 2

```
btDiscreteDynamicsWorld(btDispatcher* dispatcher, btBroadphaseInterface* pairCache,
btConstraintSolver* constraintSolver, btCollisionConfiguration* collisionConfiguration);
```

Calculates how the constraints affect the objects attached to them and "solves" how the object and it's restraints interact.

Use **btSequentialImpulseConstraintSolver**

This module contains default setup for bullet, with memory and collision setups.

For now just use **btDefaultCollisionConfiguration**

TODO 3

```
// TODO 3: Create the world (btDiscreteDynamicsWorld)
// Set default gravity
// Recommended to have gravity defined in a macro!
```

With all the pieces ready, let's create our Physics world!

#define GRAVITY as a macro, to access it from anywhere as a constant

And set your world's Gravity to your defined value.

Try to avoid objects smaller than 0.2f (Recommended 1.0f == to 1 m).

TODO 4

```
// Uncomment all the “Debug Drawer” functions
// And link them to your physics world
```

Now we can uncomment the DebugDrawer and link it to our Physics world.

It’s not really spectacular yet, though.

TODO 5

```
// TODO 5: step the world
```

Extracted from the manual (page 22):

"By default, Bullet physics simulation runs at an internal fixed framerate of 60 Hertz (0.01666). The game or application might have a different or even variable framerate. To decouple the application framerate from the simulation framerate, an automatic interpolation method is built into stepSimulation: when the application delta time, is smaller then the internal fixed timestep, Bullet will interpolate the world transform, and send the interpolated worldtransform to the btMotionState, without performing physics simulation. If the application timestep is larger then 60 hertz, more than 1 simulation step can be performed during each 'stepSimulation' call. The user can limit the maximum number of simulation steps by passing a maximum value as second argument."

TODO 6

```
// TODO 6: Create a big rectangle as ground
```

To add a rigidbody: World->addRigidBody(**btRigidBody***)

To create a rigidbody: btRigidBody(**btRigidBodyConstructionInfo**)

For construction info:

btRigidBodyConstructionInfo(**float** mass, **btMotionState***, **btCollisionShape***)

TODO 6

```
btRigidBodyConstructionInfo(float mass, btMotionState*, btCollisionShape*)
```

- **Mass**: Mass of the object. 0 for static objects, 1 by default.
- **btMotionState**: holds position, velocity, inertia... of the body. We can use **btDefaultMotionState**.
- **btCollisionShape**: defines shape of the body. We can use a big **btBoxShape**.

TODO 6

```
// TODO 6: Create a big rectangle as ground
```

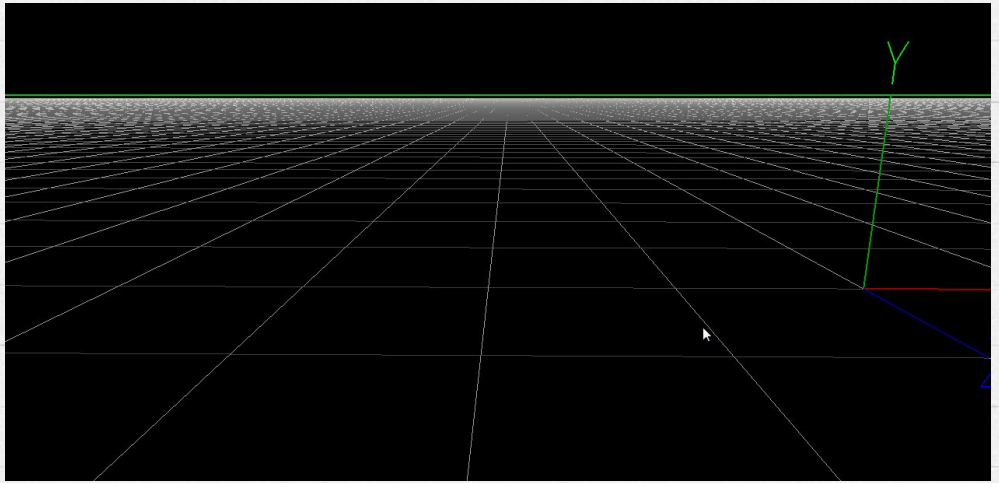
```
btMotionState *motionState = new btDefaultMotionState();
btBoxShape *shape = new btBoxShape(btVector3(200.0f, 1.0f, 200.0f));

btRigidBody::btRigidBodyConstructionInfo rigidBodyInfo(/*mass*/0.0f, motionState, shape);
btRigidBody *rigidBody = new btRigidBody(rigidBodyInfo);

world->addRigidBody(rigidBody);
```

TODO 6

We can finally see results!



TODO 7

```
// TODO 7: Create a Solid Sphere when pressing 1 on camera position
```

Similar to previous TODO, but:

- Remember to set mass to "not 0".
- We'll use a **btSphereShape**.
- We need to set a position to the body.

We mentioned **btMotionState** was holding the object's position…

TODO 7

```
// TODO 7: Create a Solid Sphere when pressing 1 on camera position
```

btMotionState holds the position, which we want to modify.

btMotionState can be constructed (or set) with a **btTransform** .

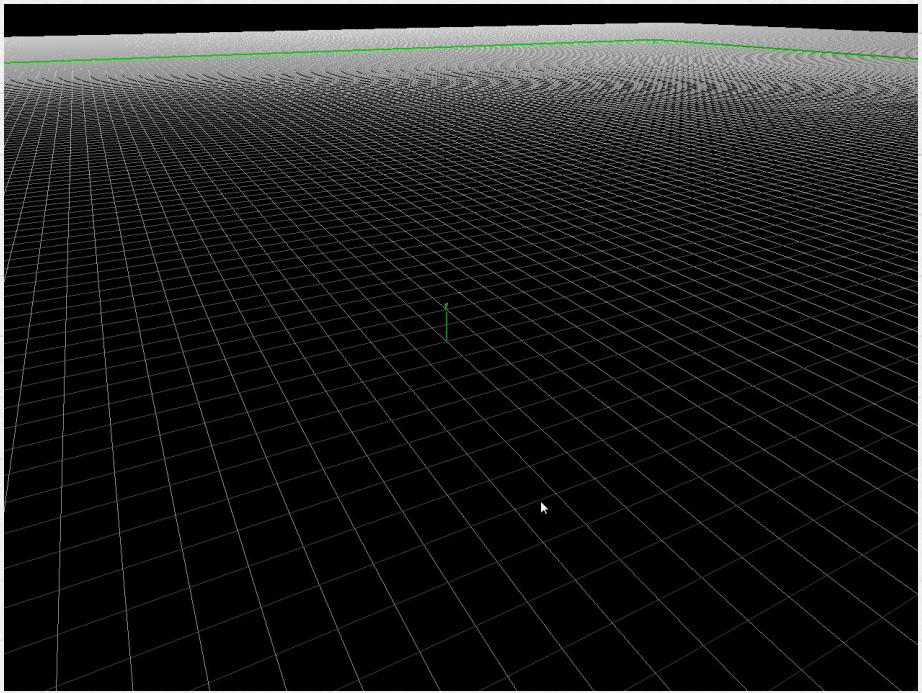
btTransform has the function setFromOpenGLMatrix(...)

glMath library allows us to create **mat4x4** (4 by 4 matrix, as used by OpenGL)

mat4x4 can be initialized to the IdentityMatrix, and translated by the **camera position**.

Where we're at

We can now spawn any physics body we want, as longs as a **btCollisionShape** exists for it!



Homework

Try to create some boxes. Experiment with different shapes!

And... think on your game! The only requirements are: car and physics.

Racing game? Mini rocket League? Obstacle course? A trailer carrying stuff?

Bring your ideas next week!

Next Week . . .

Collision Detection