

**ICT 312 Physics Simulation**

**Milestone Two**

**Design Document**

**By Arran Ford, Hamish Carrier and Timothy Veletta**

Contents

[Program Design Philosophy 3](#_Toc371516346)

[Model Information 3](#_Toc371516347)

[Program information 4](#_Toc371516348)

[Plans and Measurements 4](#_Toc371516349)

[Software Quality 5](#_Toc371516350)

[Issues 5](#_Toc371516351)

[Testing Details 5](#_Toc371516352)

[Realism 5](#_Toc371516353)

[NPC Design 5](#_Toc371516354)

[Physics 5](#_Toc371516355)

[AI 8](#_Toc371516356)

[Overall Appeal 8](#_Toc371516357)

[Special Features 8](#_Toc371516358)

[References 9](#_Toc371516359)

# Program Design Philosophy

Our approach to the design and subsequent creation of this project was a 3 step procedure. We initially started out by getting together and thoroughly planningout how we were going to approach the project. It was at this stage that we decided on what we wanted to get out of the project and what would be the best way to do it, as we were all also doing ICT313, but in separate groups, we decided that the ability to be able to work independently was one of the most important goals.

This way depending on the varying group schedules we could each work on the project at times that best fit into our own schedules. To facilitate the ability to work independently we split the first major milestone of the project up into major three categories, graphics development, collision detection and resolution, and asset creation. This kept the amount of points of overlap to a minimum, which minimised the amount of conflict we encountered when merging / sharing each of the three categories.

Once we finished the planning stage we came together as a group to measure out and record the physical design specifications of the area we planned on simulating. In addition to physical measurements we also photographed all the models we planned on creating, and any textures that we would need to replicate / make use of.

After all the physical specifications were recorded we each moved onto the category we had decided on earlier. An advantage of splitting up the tasks into three categories with as few dependencies on the other tasks as possible, was that we were able to enter the development stage and work concurrently. Without ever having to halt development while waiting on another party to finish a part of their section.

As a result of our design philosophy we were able to develop much quicker than we had done in previous group projects, and the number of problems we encountered was also much lower than previously experienced.

## Model Information

All of the assets were created in two programs, the 3D models in 3DS Max, and the textures in Photoshop. This was done due to the ease of use of the programs, and due to the ability of Ogre3D to easily render pre-made 3D files. The models were exported from 3DS Max to Ogre3D scene, mesh and material files using the OgreMax plugin for 3DS Max.

The models are loaded into our program using the ‘SceneLoader’ class which loads in the .scene format produced by OgreMax. The scene file is essentially an XML file that defines each of the models in the scene, their position, orientation and scale and also the material associated with that model.

The scene loader then creates new ‘GenericObjects’ from each of the models loaded in and adds them to the current scene.

## Program information

Our program consists of 5 main parts, classes associated with running the game, classes associated with rendering, the collision system, the physics system and the AI system.

The ‘Game’ class brings all the other sections together and contains the program loop for updating the program.

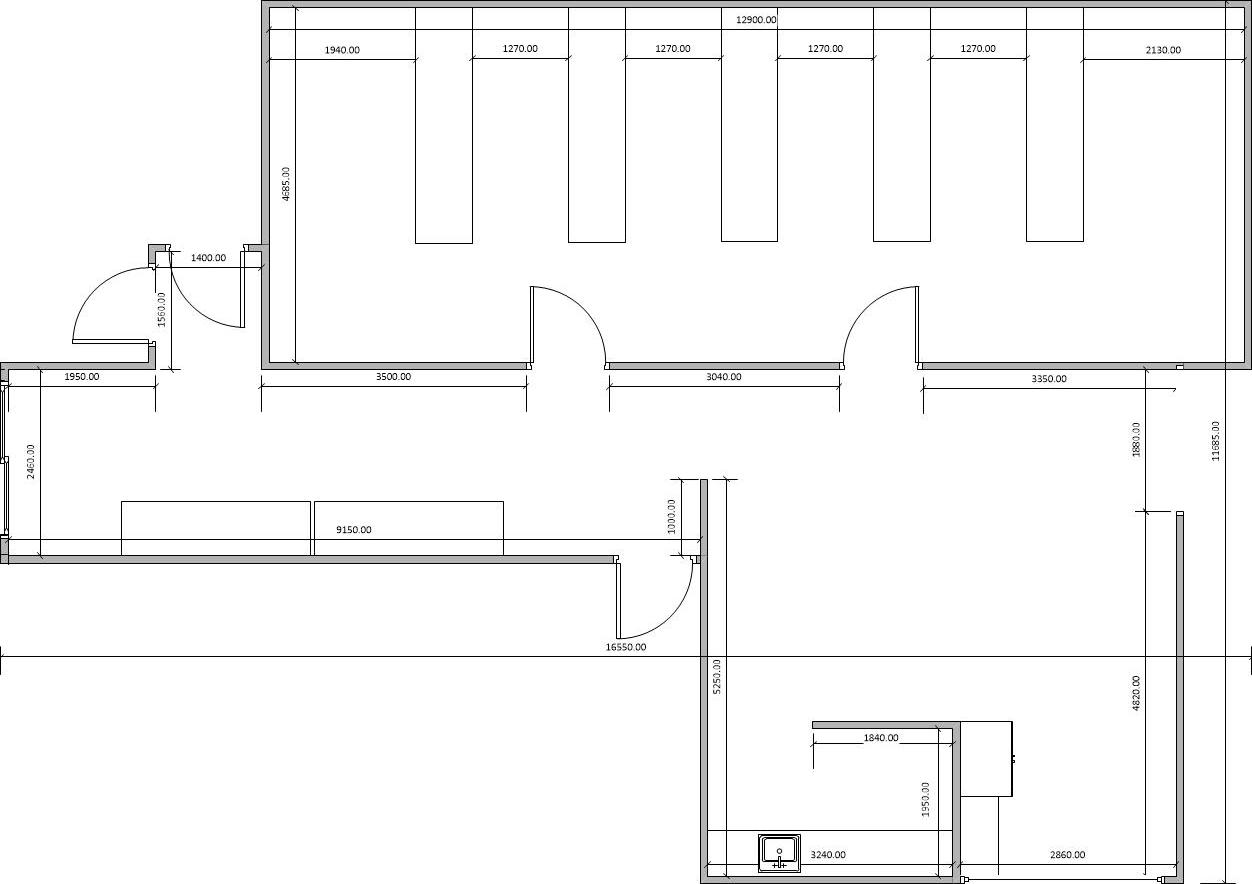
For rendering we use the Graphics API ‘Ogre3D’ with all the functionality we require defined in functions in the ‘OgreGraphics’ class. All of the components in the Graphics namespace can be used independent of the other parts of the system so they can be reused in latter projects.

The collision system uses the API “Bullet Physics” to detect collisions, each collision object in the world has a void pointer that points to a custom object we create that is associated with the collision object, allowing the collision system to trigger collision resolution in the objects that have collided.

The physics system uses an impulse based system for resolving collisions and uses Verlet integration for determining an objects change in position and velocity over time. The reasoning for using Verlet integration over Euler integration for determining object positioning is because Verlet offers a greater level of accuracy since it uses the average acceleration between frames to determine the objects velocity. The impulse based physics system is detailed in a latter section.

The AI system…

## Plans and Measurements



# Software Quality

Some of the things we tried to do to ensure the quality of the software included:

* Enforcing of the coding standards defined in our Team Charter.

## Issues

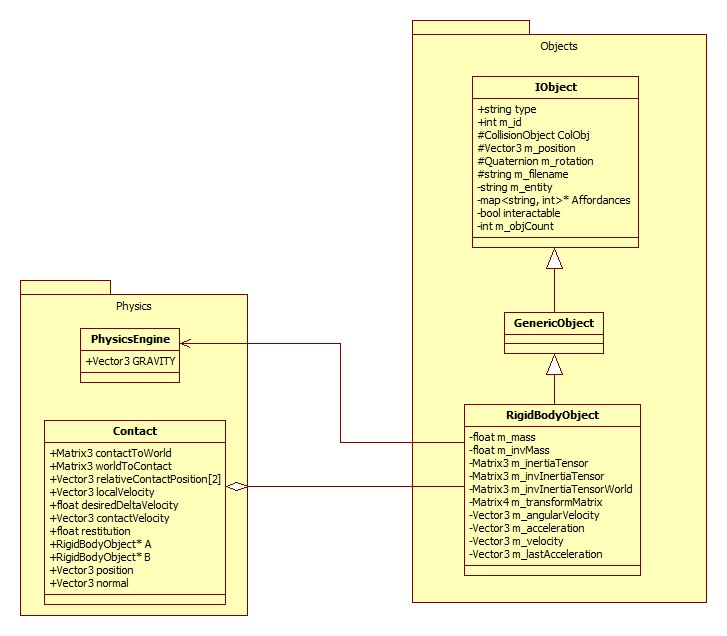
## Testing Details

# Realism

## NPC Design

## Physics

The Physics system is primarily confined to classes within the Physics namespace as well as the ‘RigidBodyObject’ class which is part of the Objects namespace.



The process used to resolve collisions within our engine is as follows.

1. **When a collision is detected by the collision world, create an instance of ‘Contact’ passing in the two objects involved in the collision as well as the point of collision and collision normal.**

This is primarily handled in the CollisionWorldSingleton class:

if((obB->getUserPointer())&&(obA->getUserPointer()))

{

mani.A = static\_cast<Objects::RigidBodyObject\*>(obA->getUserPointer());

mani.B = static\_cast<Objects::RigidBodyObject\*>(obB->getUserPointer());

if(contactManifold->getNumContacts() > 0)

{

btManifoldPoint contact = contactManifold->getContactPoint(0);

Physics::Contact(

static\_cast<Objects::RigidBodyObject\*>(obA->getUserPointer()), static\_cast<Objects::RigidBodyObject\*>(obB->getUserPointer()), Ogre::Vector3(contact.getPositionWorldOnA().getX(),

contact.getPositionWorldOnA().getY(),

contact.getPositionWorldOnA().getZ()),

Ogre::Vector3(contact.m\_normalWorldOnB.getX(),

contact.m\_normalWorldOnB.getY(),

contact.m\_normalWorldOnB.getZ()));

}

}

Resolving the collision is then left to the Contact class within the Physics namespace.

1. **Convert the position of the objects into a coordinate system local to the point of contact and the contact normal. A transform matrix is created to convert to and from this coordinate system.**

In this stage we are mainly concerned with changing positions from world coordinates to coordinates local to the collision contact point. This is to simplify some of the calculations further on in the process.

This is handled in the ‘CalculateBasis’ function within the Contact class which creates a set of axes with the x-axis pointing down the Contact normal. Since the x-axis has been determined, an arbitrary y- and z-axis must be defined.

The ‘CalculateBasis’ function calculates the axes and then creates a set of Matrices that allow for conversion to and from this set of axes.

1. **The change in velocity per unit impulse of the contact point on each object is worked out. This value needs to take into account both the linear and angular motion.**

Since the physics engine deals with frictionless contacts, the only impulses generated at the contact are applied through the contact normal. The goal of this step is to come up a value for the amount the velocity changes relative to the contact, in the direction of the contact normal per unit impulse applied.

This value will have both a linear and an angular component which can be dealt with separately and combined at the end.

The linear change in velocity per unit impulse will be in the direction of the impulse with the magnitude given by the inverse mass of the object. For collisions, the linear component is simply the sum of the two inverse masses. i.e.

The linear change in velocity is handled in the ‘CalculateImpulse’ function in the Contact class as follows:

deltaVelocity += A->getInverseMass();

deltaVelocity += B->getInverseMass();

The angular change in velocity per unit impulse is slightly more complex; we start by finding the point of contact relative to the origin of both objects. This value is then crossed with the contact normal to work out the amount of impulsive torque generated per unit impulse. i.e.

This value is then multiplied by the inverse inertia tensor of the object to obtain the change in angular velocity per unit of impulsive torque. This is handled in code in the ‘CalculateImpulse’ function in the Contact class as follows:

Ogre::Vector3 deltaVelWorld = relativeContactPosition[0].crossProduct(normal);

deltaVelWorld = inverseInertiaTensor[0] \* deltaVelWorld;

deltaVelWorld = deltaVelWorld.crossProduct(relativeContactPosition[0]);

The ‘CalculateImpulse’ function described above essentially calculates the lower part of the equation from Rabin’s book, Introduction to Game Development. i.e.

1. **Invert the previous stage to find the impulse needed to generate a velocity change.**

For frictionless collisions, if we have a single value for velocity change per unit impulse (*d*) then the impulse needed to produce a given velocity change is specified by:

Where v is the desired change in velocity and *g* is the impulse required.

1. **Work out the separation velocity, the current closing velocity and then the difference between the two which is stored as the desired change in velocity.**

First to calculate the current closing velocity using both the linear and angular component which is done in the ‘CalculateLocalVelocity’ function within the Contact class. i.e.

velocity = A->getRotation().crossProduct(relativeContactPosition[0]);

velocity += A->getVelocity();

This value is then made more accurate by taking the velocity due to acceleration in the previous update step into account through the line:

accVelocity = A->getLastAcceleration() \* Core::Game::getGraphics()->getDeltaTime();

The current closing velocity for both objects involved in the collision is combined in the ‘Initialise’ function of the contact class. i.e.

contactVelocity = CalculateLocalVelocity(true) - CalculateLocalVelocity(false);

The desired velocity change in then calculated by combining the current closing velocity with the restitution of the two objects however the velocity due to acceleration in the previous update step must also be taken into account and removed to ensure an accurate velocity post collision. i.e.

desiredDeltaVelocity = -contactVelocity.x - restitution \* (contactVelocity.x - velocityFromAcc);

Throughout this step we have calculated the upper part of the equation from Rabin’s book, Introduction to Game Development. i.e.

1. **Using the desired change in velocity calculate the impulse that must be generated.**

Since we are only concerned with the impulse in the direction of the contact normal, we will only be concerned with the x-axis of the final impulse vector which is in contact coordinates. This is implemented through:

impulseContact.x = desiredDeltaVelocity / deltaVelocity;

impulseContact.y = 0;

impulseContact.z = 0;

This value can then be converted back into the world coordinate system to be applied to the objects involved in the collision.

1. **Split the impulse into linear and angular components and apply them to each object.**

The linear velocity change is simply calculated by multiplying the impulse by the inverse mass of the body as follows:

velocityChange[0] = impulse \* A->getInverseMass();

velocityChange[1] = impulse \* -1 \* B->getInverseMass();

The change in rotation is given by the following formula:

Where is the change in rotation, is the inverse inertia tensor of the object, *r* is the relative contact position and *g* is the impulse. This is done through:

Ogre::Vector3 impulsiveTorque = relativeContactPosition[0].crossProduct(impulse);

rotationChange[0] = inverseInertiaTensor[0] \* impulsiveTorque;

The linear and angular velocity changes are then applied to the objects involved in the collision through:

A->addVelocity(velocityChange[0]);

A->addRotation(rotationChange[0]);

B->addVelocity(velocityChange[1]);

B->addRotation(rotationChange[1]);

## AI

# Overall Appeal

Explain how the software presents when running. Describe how stable is the software.

# Special Features

Special features3 – like networking, Player – NPC dialog or anything else you put in. Explain these features.

Illustrate does not mean that you just copy and paste code. An explanation is needed and the code is to backup the explanation.

# References

Contain reference to Debug Drawers here