

**ICT 312 Physics Simulation**

**Milestone Two**

**Design Document**

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

Contents

[Program Design Philosophy 3](#_Toc371061360)

[Model Information 3](#_Toc371061361)

[Program information 4](#_Toc371061362)

[Plans and Measurements 4](#_Toc371061363)

[Software Quality 5](#_Toc371061364)

[Issues 5](#_Toc371061365)

[Testing Details 5](#_Toc371061366)

[Realism 5](#_Toc371061367)

[NPC Design 5](#_Toc371061368)

[Physics 5](#_Toc371061369)

[AI Used 5](#_Toc371061370)

[Overall Appeal 5](#_Toc371061371)

[Special Features 6](#_Toc371061372)

[References 6](#_Toc371061373)

# 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 (and in two different 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.

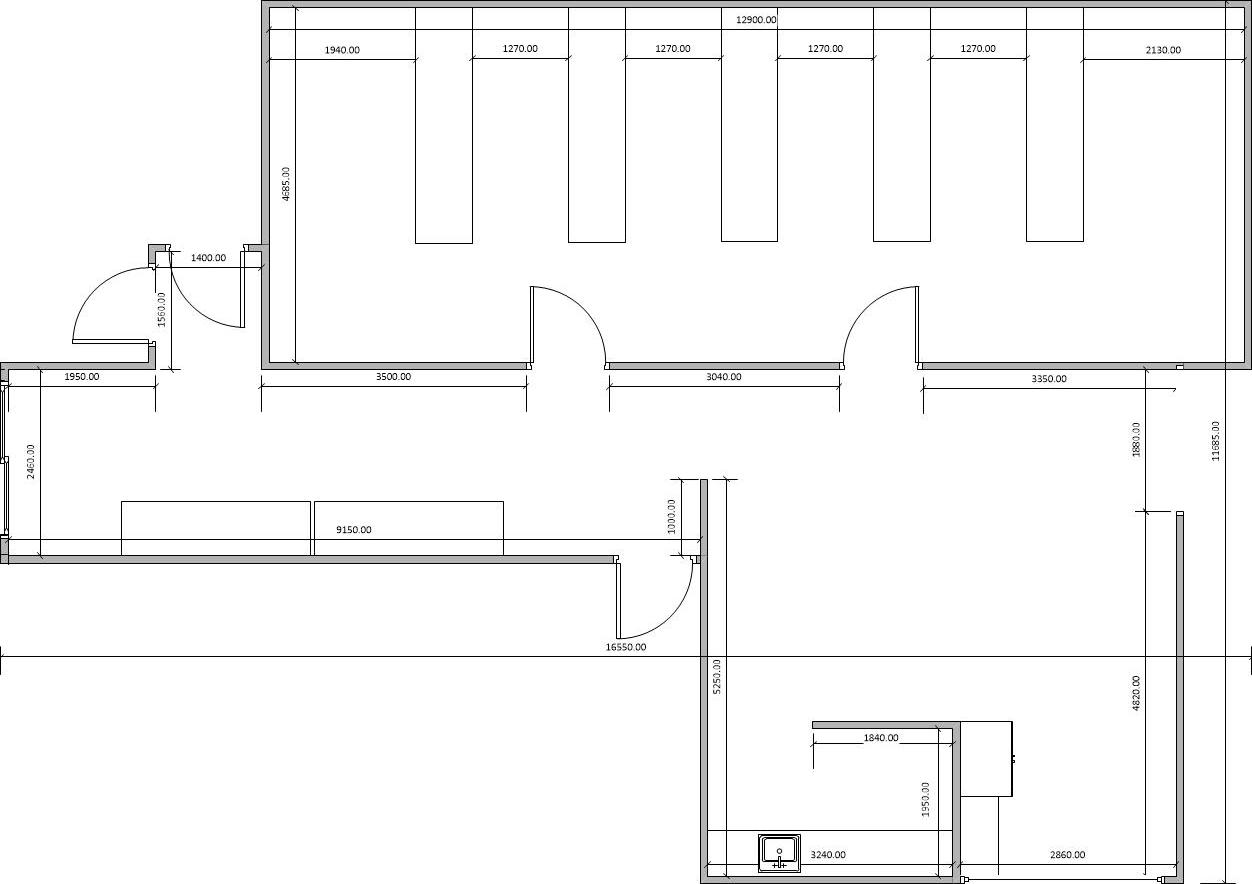
For rendering we use the Graphics API ‘Ogre3D’ with all the functionality we require defined in functions in the ‘OgreGraphics’ class. All of the components in the Graphics namespace can be used independent of the other parts of the system so they can be reused in latter projects.

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. To build complex objects requires making a “compound object” made up of primitive shapes as the bullet API does not support mesh to mesh collisions.

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

## Issues

We encountered many issues during the development of this project our primary one was not related to software and was instead managing our final semester workloads, this caused the time we spent on the project and our communication to be disjointed, which in turn caused slow development of our program.

In Software we each experienced many issues in our respective areas.

Collision: The biggest issue in this area was caused by attempting to use “Ogre Bullet” which is a set of classes that attempt to simplify using the Bullet Physics API with Ogre, however this library was extremely poorly documented which lead to a lot of trial and error and difficult to solve problems such as collisions not being detected in the world. Eventually we have up on Ogre Bullet as we were unable to get it working and instead used raw Bullet. Bullet had its own host of issues the two primary ones being the debug draw functionality and mesh to mesh collisions. The debug draw function simply draws every line in the world one after the other with no use of data structures or other methods to optimise it, this caused to a huge drop in frame rate every time the drawer was enabled, given that we were already behind in the project our solution was to find a slightly more efficient drawer and gave the draw function the ability to be toggled on and off this allowed us to view meshes and then turn off the drawer to navigate the world. Mesh to mesh collisions in bullet physics are not supported however the documentation gave the impression that collision did work with convex meshes, however this was proved to be false which left us with no way to detect collisions on complex shapes. Our solution was to build complex shapes out of primitive shapes and combine them into a bullet “compound shape” we didn’t have a chance to implement this however.

AI has had several issues throughout the development lifecycle. The first is the use of boost::statechart, although it works fine in isolation and handles a lot of things such as state transitions and state creation/destruction there were issues with getting it integrated and working with slightly more complex state behaviours. For this reason, it was scrapped and replaced, although the new system is fundamentally just as simple it is much more expansible for future state additions.

The AI also had issues with integration with the affordance system, it was at one stage unable to handle multiple affordance types. This was a simple issue that was resolved with cooperation between the two participating team members.

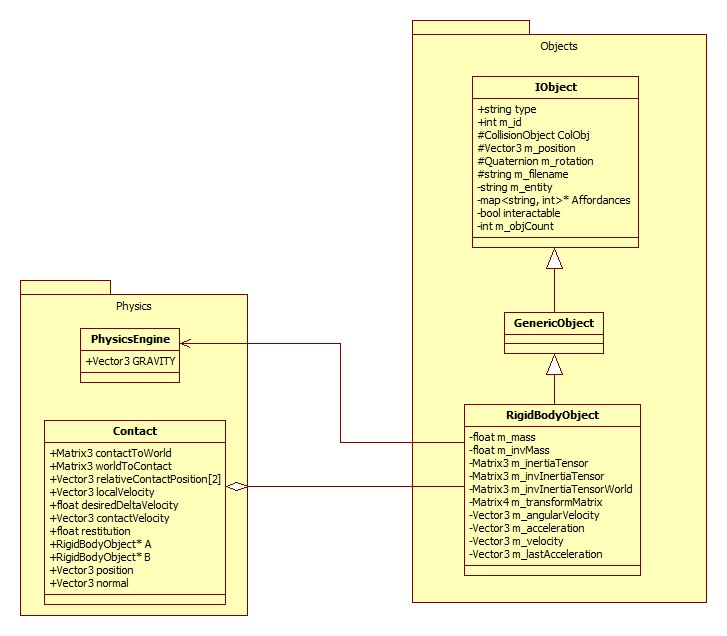
**Tim issues here**

## Testing Details

# Realism

## 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 Used and Design

When creating the AI we one of the first things we did was figure out which parts of a real character were viable to be simulated, and how important they were in creating something that was realistic. We ended up with a list of the things that were important, and implementable. It looked as follows;

Thinking – We wanted the AI to be able to make its own decisions based on its environment and its own state

Feeling – We wanted our AI to be able to be affected by its actions and its environment

Interacting – We wanted our AI to be able to impact the environment

Communicating – We wanted our AI to be able to be affected by the player character

We ended up having to cut Interacting, and dumbing down Communicating in our final deliverable due to time and other constraints.

**Thinking**

To achieve a thinking AI we used primarily two things; a simple FSM (originally boost::statechart, but switched later due to complications) which was in charge of controlling the overall NPC behaviour, and a deterministic goal selection function which attempts to simulate a nondeterministic one through the use of randomness.

The FSM is structured as follows;

bool NPC::runCurrentState()

{

switch(CurrentState)

{

case EnumSpace::enumThinking:

{

\**THINKING CODE\**

}

case EnumSpace::enumIdling:

{

\**IDLING CODE\**

}

case EnumSpace::enumInteracting:

{

\**INTERACTING CODE\**

}

case EnumSpace::enumSearching:

{

\**SEARCHING CODE\**

}

}

}

As you can see it is made up of four primary states, Thinking, Idling, Interacting and Searching. These states are controlled and switched between by the use of the CurrentState variable, which tracks which state is active. We will now explore each of these states in more detail.

case EnumSpace::enumIdling:

{

std::cout << "Idling" << std::endl;

if(rand() %2 == 0)

{

std::cout << "Motivated, examining goals..";

CurrentState = EnumSpace::enumThinking;

return true;

}

else

{

return false;

}

break;

}

The first and default state is Idling, in this state the NPC only does two things; One; it prints out its current state, and Two; it checks whether or not the NPC should become ‘motivated’ enough to begin thinking about goals and considering its own status. This is simulated simply by a 50/50 random chance.

case EnumSpace::enumThinking:

{

std::cout << "Thinking" << std::endl;

if(DetermineGoal())

{

std::cout << "Goal Determined" << std::endl;

CurrentState = EnumSpace::enumSearching;

return true;

}

else

{

std::cout << "Demotivated, no goal selected" << std::endl;

CurrentState = EnumSpace::enumIdling;

return false;

}

break;

}

In this state the NPC will activate its goal determination function (this will be explored further down), if it successfully selects a goal then it will state so and move onto searching for an appropriate object in the environment that will allow for completion of the goal. If not it will become demotivated and return to idling, just as a person who can’t determine a goal will do nothing important.

case EnumSpace::enumSearching:

{

std::cout << "Searching for interactable object... ";

ObjectPointer = NULL;

ObjectPointer = ItemStore::Instance()->GetObject(myObj->getPosition(), "Sit", CurrentGoal->GetThreshold());

if ( ObjectPointer == NULL )

{

CurrentGoal->DecayThreshold();

}

else

{

std::cout << "Found an object with suitable affordance";

CurrentState = EnumSpace::enumInteracting;

}

return true;

break;

}

This state handles the NPCs ability to search for an appropriate object in the world to do this it resets its current object interaction pointer. It then checks the item store to see if there are any items available with the relevant criteria (the current NPC position, the type of action and the affordance value required). If there isn’t the NPC will lower its expectations (just as someone would in real life if they had a desire unfilled over time) and look again for anything that will make do. Once an item is found it is stored by the NPC and the state is progressed to interacting.

case EnumSpace::enumInteracting:

{

std::cout << "Interacting" << std::endl;

ObjectPointer->SetInteractable(false);

if(CurrentGoal->GetAction()->Activate())

{

EmotionCheck();

ObjectPointer->SetInteractable(true);

CurrentState = EnumSpace::enumIdling;

}

else

{

CurrentNeeds[EnumSpace::enumFun] -= (\*CurrentGoal->ModifyNeeds())[EnumSpace::enumFun];

CurrentNeeds[EnumSpace::enumGrades] -= (\*CurrentGoal->ModifyNeeds())[EnumSpace::enumGrades];

CurrentNeeds[EnumSpace::enumComfort] -= (\*CurrentGoal->ModifyNeeds())[EnumSpace::enumComfort];

if(CurrentNeeds[EnumSpace::enumFun] < 0)

{

CurrentNeeds[EnumSpace::enumFun] = 0;

}

if(CurrentNeeds[EnumSpace::enumGrades] < 0)

{

CurrentNeeds[EnumSpace::enumGrades] = 0;

}

if(CurrentNeeds[EnumSpace::enumComfort] < 0)

{

CurrentNeeds[EnumSpace::enumComfort] = 0;

}

}

return false;

break;

}

The final state is interacting, in this state the NPC flags the item it is currently using as being busy (this way other NPCs will not be able to also fetch the same busy item when looking for their own interactable item). The NPC then activates the action attached to its current goal, if it returns true then the goal has been completed and the state and all relevant information is reset. If not the NPCs needs are affected by the current type of action and goal being performed, this is to simulate the kind of real life feedback that people get from performing actions.

**Goal Determination Function**

To simplify this will be broken down into parts;

int Grades = 0;

int Comfort = 0;

int Fun = 0;

Grades = CurrentNeeds[EnumSpace::enumGrades] + myTrait->GetGoalModifier()[EnumSpace::enumGrades];

Comfort = CurrentNeeds[EnumSpace::enumComfort] + myTrait->GetGoalModifier()[EnumSpace::enumComfort];

Fun = CurrentNeeds[EnumSpace::enumFun] + myTrait->GetGoalModifier()[EnumSpace::enumFun];

int Total = Grades + Comfort + Fun;

int Outcome = 0;

if(Total != 0)

{

Outcome = rand() % Total;

}

int NeedLevel = 0;

EnumSpace::NeedTypes Need;

//getchar();

if(Outcome > (Total -= Fun))

{

std::cout << "Prioritising Fun" << std::endl;

NeedLevel = Fun;

Need = EnumSpace::enumFun;

}

else if(Outcome > (Total -= Comfort))

{

std::cout << "Prioritising Comfort" << std::endl;

NeedLevel = Comfort;

Need = EnumSpace::enumComfort;

}

else if(Outcome > (Total -= Grades))

{

std::cout << "Prioritising Grades" << std::endl;

NeedLevel = Grades;

Need = EnumSpace::enumGrades;

}

This section is used to simulate what type of goal the NPC is most predisposed to select. This is affected by the type of traits the NPC has. Just in real life certain types of people are more likely to perform certain actions, so are the NPC. Here the NPC simply adds all of its goal priorities and randomly selects one based on their distribution.

Outcome = rand() % 100;

if(CurrentMood->GetType() == EnumSpace::enumGood)

{

Outcome - 10;

}

else if(CurrentMood->GetType() == EnumSpace::enumBad)

{

Outcome + 10;

}

if(Outcome < NeedLevel)

{

std::cout << "Picking a goal" << std::endl;

switch(Need)

{

case EnumSpace::enumFun:

{

CurrentGoal = new Relax;

break;

}

case EnumSpace::enumComfort:

{

CurrentGoal = new Relax;

break;

}

case EnumSpace::enumGrades:

{

CurrentGoal = new Work;

break;

}

}

CurrentGoal->ResetThreshhold();

return true;

}

The next section determines whether or not the NPC has a need strong enough, or a desire strong enough to enact on its choice. This is simulated in two parts; the first is the mood, a good mood increases the NPCs motivation to select a goal, and a bad decreases it. This simulates real life in that people feeling good are more likely to be productive, as opposed to people in a bad mood being unmotivated. The second part is checking the current need, as the need level is increased it becomes more and more likely that a relevant goal will be picked. This is to simulate the increase in the urgency of a need to fulfil a need over time.

**Feeling**

We broke this section of the AI down into several components;

**Needs**

This follows a very popular existing model (most commonly known from The Sims), where by the NPC has a set of needs that increase overtime, and are decreased through specific activities. This system allows for the AI to take into consideration what goals are best suited to fulfil its current needs, and helps to increase the variety of actions chosen, as an unfulfilled need will slowly become higher and higher priority. The code for this is shown in the Goal Determination Function above.

**Emotions**

These are implemented as a set of classes, managed by a main controller class; Emotion Manager.

ProgressMood(EmotionalOutcome);

Each emotion triggered impacts the mood, once enough of one emotion has been triggered the mood will change (the change depends on the emotion first reached). Then the counter is reset to nill.

EnumSpace::EmotionTypes EmotionalOutcome = EnumSpace::enumNeutral;

The default emotional outcome is set to neutral (which is the same as no outcome as it has no effect on moods).

EmotionalProb = rand() % Total;

for(int i = 0; i < EnumSpace::EmotionTypes\_Max; i++)

{

if(EmotionalProb <= (X += TotalModifiers[i]))

{

EmotionalOutcome = (EnumSpace::EmotionTypes)i;

}

}

A random number within the bounds of the total modifiers is chosen, it is then checked against the distribution of each emotional modifier. If the random number falls within that distribution, the outcome is set to the emotion relevant to that emotional modifier. This system was chosen to be able to setup modifiers for as many different things as needed (in this case moods and actions). The system is also flexible in that if additional emotions are added, no other changes are necessary in the computational functions.

**Moods**

These are implemented as a set of classes, managed by a main controller class; Mood Manager. Each mood has an impact on the probability of an emotion occurring.

int Multipliers[] = {1, 3, 0};

InitialiseMood(EnumSpace::enumGood, Multipliers);

In this above example, a Good mood is initialised to have a probability increase of 1 for a neutral emotion, 3 for a happy emotion and 0 for a sad.

TotalModifiers[i] += (MoodManager::GetInstance()->GetEmotionMultipliers(CurrentMood->GetType())[(EnumSpace::EmotionTypes)i]);

This code shows where the modifiers are used; they are combined in this function with all other modifiers which is then later applied to selecting an emotion.

**Communicating**

Although we were running short of time in implementing this we felt that this was still an important enough aspect to realistically simulating AI that we didn’t want to cut it completely. What we ended up with was a very simple way to interact with the AI, through both positive and negative feedback from the player. Our simulation allows the player to trigger emotions in the AI through the use of the mouse (left click for sadness, right click for happiness).

The function that implements this is the following, and is very simple.

void NPC::Clicked(std::string Click)

{

if(Click == "Left")

{

std::cout << "Hey don't click me!";

ProgressMood(EnumSpace::enumSad);

}

else

{

std::cout<<"That feels nice!";

ProgressMood(EnumSpace::enumHappy);

}

}

Here we take in a type of click and output the NPC response and progress their current mood as if a happy or sad emotion was triggered.

# Overall Appeal

Our simulation is quite visually appealing, all textures have been hand made from photos of the room we modelled and all models are also handmade the texturing of our roof was rushed however and the photos that depict the edges of our simulation could be better.

As for stability the program does run stably though the frame rate is quite low and is significantly reduced by the debug drawer.

# Special Features

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

Our project does contain some special features.

The first of these Is path finding functionality, while not implemented in our simulation we do have the ability to create Astar maps and to return paths to the nearest node to a point. We have done this using the Grinning Lizard micropather which is an A\* solver that when implemented with a map and functions to calculate costs between nodes can return a path of those nodes.

mapper = new WorldMap;

mapper->FindPath(Ogre::Vector3(1,0,0),Ogre::Vector3(2,0,10));

MapNode \* temp;

for(int i = 0; i<mapper->path.size(); i++)

{

temp = (MapNode\*)mapper->path[i];

cout<<"Location of "<<i<<" node ="<< temp->GetLocation().x<<temp->GetLocation().y<<temp->GetLocation().z<<"\n";

}

These lines of code show the basic use of our pathfinding, an object of type WorldMap is created and then a start and end point Is passed into the find path function.

This will then fill the path vector in the worldmap object with the list of nodes that need to be traversed to reach the nearest node to the destination. Nodes are currently hardcoded into the class NodeContainerSingleton, each node contains a position and a vector of pointers to the nodes that can be moved to from it. Costs between nodes simply use squared distances.

The WorldMap class is an implementation of the abstract class provided by the Grinning lizard micropather.

Another special feature is our ability to click on NPCs to induce a response in them. This is done using a ray casting function called “TestSelect” that is defined in Game.h. Test select is a void function that creates a ray from the player’s position along their view vector using ogre functions

Ogre::Ray mouseRay(Core::Game::getGraphics()->GetPosition(),Core::Game::getGraphics()->cameraDirection());

mRaySceneQuery->setRay(mouseRay);

mRaySceneQuery->setSortByDistance(true);

mRaySceneQuery->setQueryMask(Targetable);

// Execute query

Ogre::RaySceneQueryResult &result = mRaySceneQuery->execute();

The query mask function ensures the ray only returns objects that have been flagged as targetable, in this case any object that could be an NPC. We then iterate through the returned objects.

for(rayIterator = result.begin(); rayIterator != result.end(); rayIterator++ )

{

if ((\*rayIterator).movable !=NULL && closestDistance>(\*rayIterator).distance && (\*rayIterator).movable->getMovableType() != "TerrainMipMap"&& (\*rayIterator ).movable->getName() != "entity1" && (\*rayIterator).movable->getQueryFlags() == Targetable)

until we find the closest object that fits our parameters. The object we find is an ogre entity which contains a pointer to our custom object that the ogre entity represents when we render. We then use the NPC ID stored in this object if there is one to trigger a happy or sad emotional response in the AI that was clicked on depending on whether the ray function was triggered by a right or left click.

if(temp->AI > -1)

{

Controller->GetNPC(temp->AI)->Clicked(click);

Where “click” is of type string and was set at the start of the function depending on the type of click.

if(Click == "Left")

{

std::cout << "Hey don't click me!";

ProgressMood(EnumSpace::enumSad);

}

else

{

std::cout<<"That feels nice!";

ProgressMood(EnumSpace::enumHappy);

}

This is the clicked function within the NPC that creates a response based on the click type.

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

# 

# References

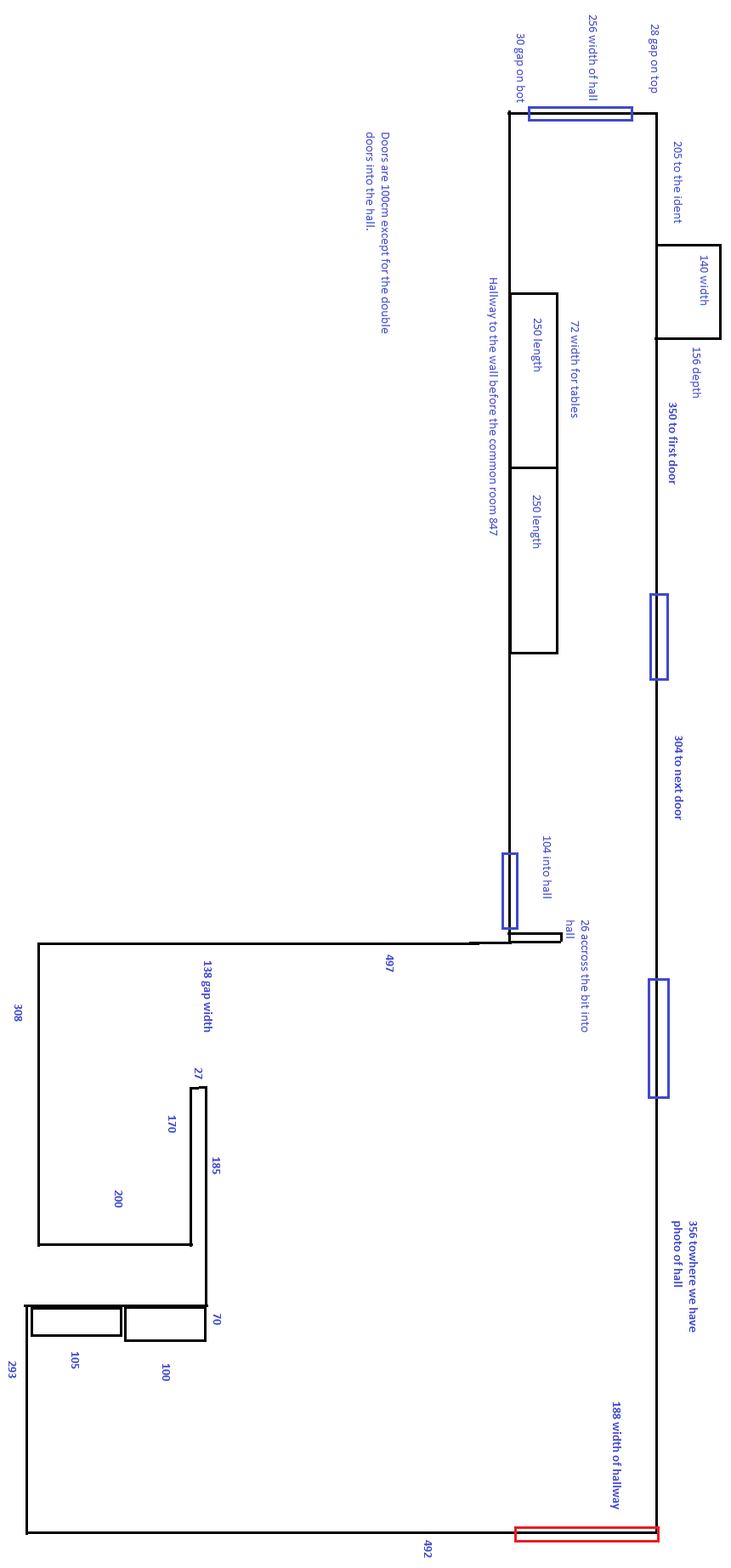
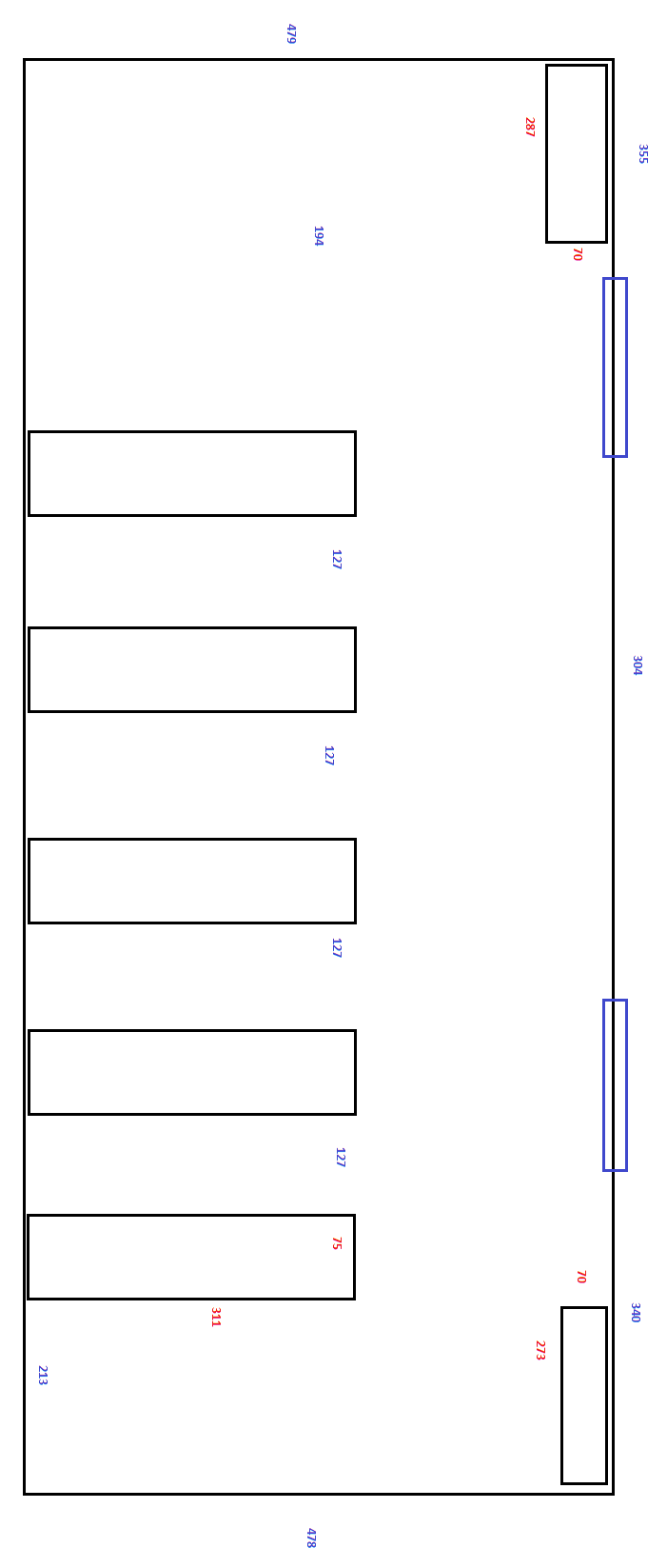
The class DebugDrawerOg is an implantation of an abstract Bullet degub draw class that was taken from the Ogre bullet wiki. http://www.ogre3d.org/tikiwiki/BulletDebugDrawer&structure=Cookbook

The Raycasting function TestSelect is based off Ogre tutorials.

The Micropather.cpp and Micropather.h files are part of the Grinning lizard A\* solver.

<http://www.grinninglizard.com/MicroPather/>

Contain reference to Debug Drawers here



For larger versions of images see:

<http://i.imgur.com/H0yi7Zr.png>

and

<http://i.imgur.com/I7tqwMN.png>

respectively.