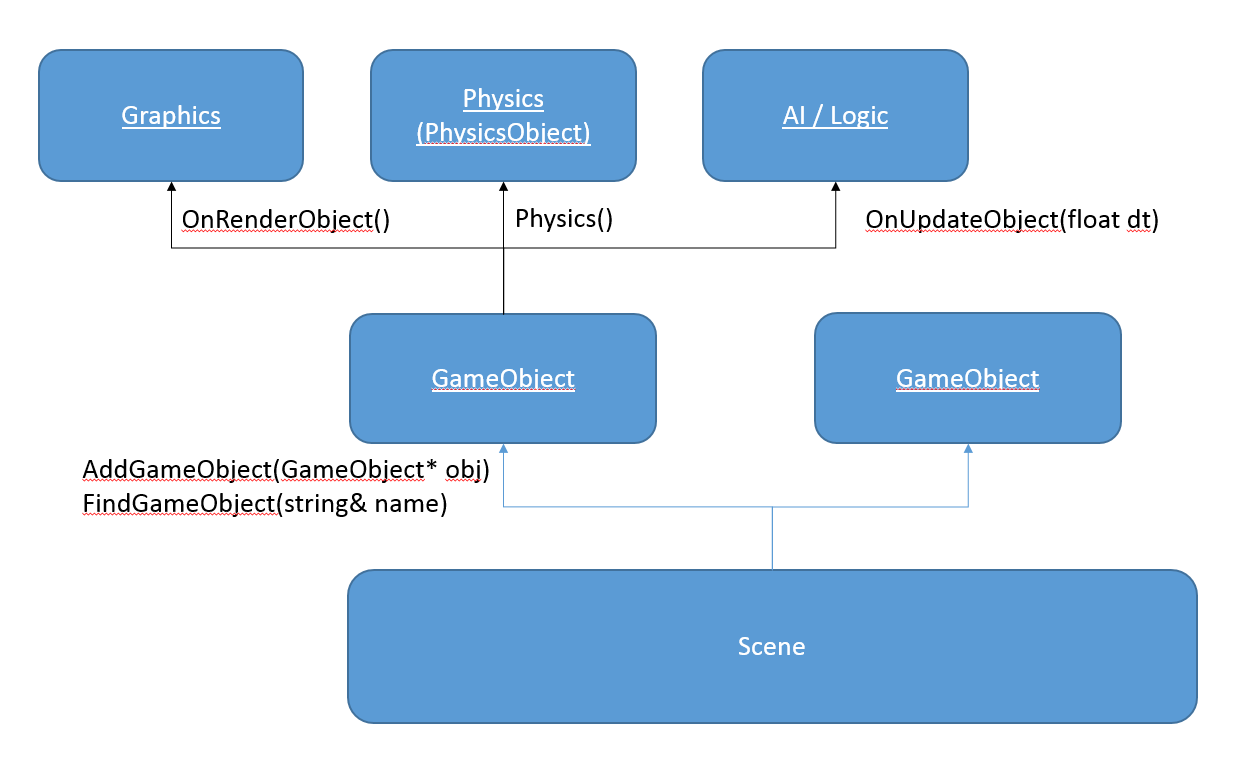
NCLTech

## Introduction

Welcome to the new NCLTech framework. Over the past month you will have become familiar with rendering with the nclgl framework. However, we needed to take a step to go from graphics orientated programming to more generic game engine. The framework provided, which is free for you to adapt as you wish, has been setup in order to segregate AI/Physics and Graphics.

## Framework Structure

The framework has been laid out as such:



Where scene represents your game world, containing, updating and rendering all your game objects in turn. Which should move a lot of the code to the specific GameObjects you create (No more ~2000 line renderer.cpp files).

Each GameObject represents a single object in the world (with possible child objects) and can optionally be given a text name in order to be located with *“FindGameObject”* later. It has an overridable *“OnUpdateObject(float dt)”* and *“OnRenderObject()”* functions to catch update and render requests as they occur.

The Graphics still uses NCLGL, with meshes and all the fun stuff that entails. However, as this is a game technologies module and not a graphics module, the scene has a basic renderer and shader built in that will automatically render any meshes drawn with the “*(mesh\*)->draw();”* function.

The AI, along with game logic, can now be implemented within the “OnUpdateObject(float dt)” function of each game object. There is also useful functionality available to help interaction between objects.

These functions are:

**m\_Scene->FindGameObject(<string name>);**

You can find other game object’s in your scene by name, returning either the game object or NULL. If no object could be found. N.B. this is currently just a depth first search of all objects so is not the fastest thing to call all the time.

**Physics()->SetOnCollisionCallback(<function>);**

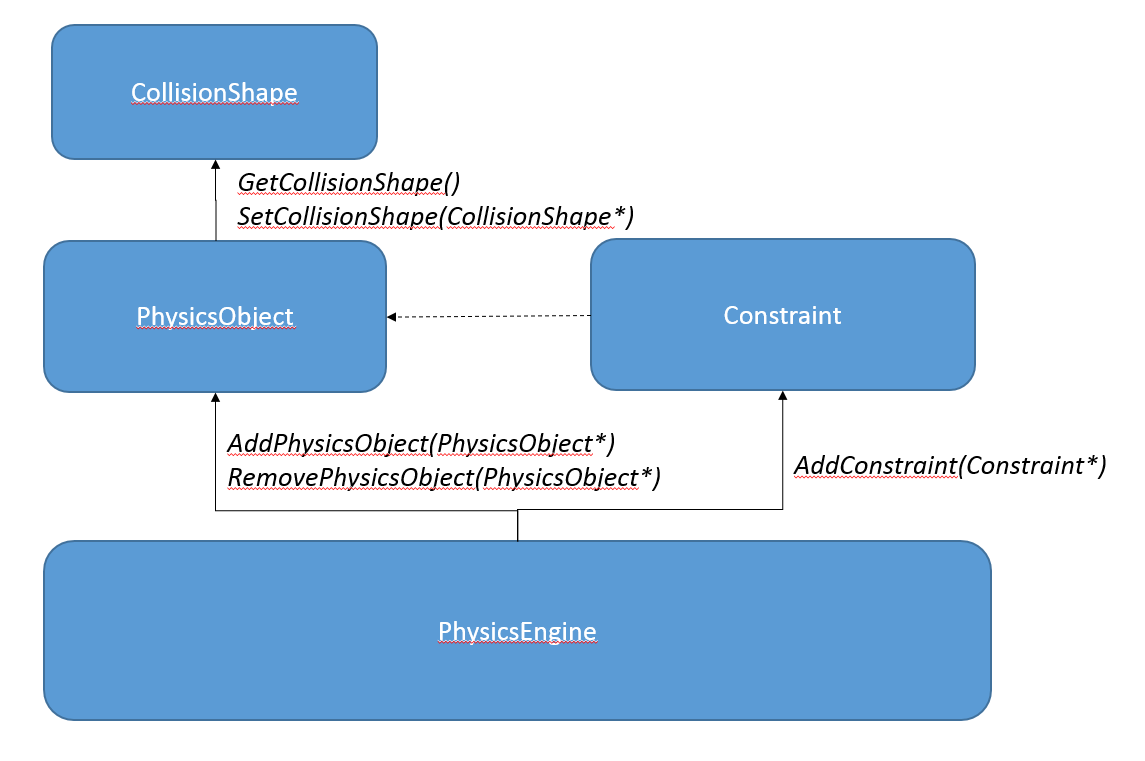
The physics system has a collision callback that will be called the moment an object collides with another object. The callback should return true to carry out full collision resolution or false to ignore. The function is in the form of:

*bool myCollisionCallback(PhysicsObject\* object\_im\_colliding\_with);*

Lastly the physics (which will be explained in more detail) is linked to each game object with a PhysicsObject, which can be both retrieved and altered with “(\*)->*Physics()”.* This will contain all physical information (and collision data) about the game object and will update the world transform of the object each physics timestep.

## Physics Structure

The physics structure is laid out as such:



Where PhysicsEngine is a Singleton entity that can be retrieved and accessed from anywhere in the program with *“PhysicsEngine::Instance()”.*

The PhysicsEngine class is in charge of all core aspects of the physics pipeline (broadphase, narrowphase, updates and constraint solving). With PhysicsObjects and Constraints (constraining two objects) forming the entities that the engine utilises.

A separation has been made between the PhysicsObject and the CollisionShape to allow the CollisionShape implementation to vary to model various shapes. These can be currently seen in CuboidCollisionShape and SphereCollisionShape.

## NCLDebug

One core new component to the framework, which should be useful for debugging, profiling and visualising geometric algorithms is NCLDebug.

This static class can be called from anywhere in the program and will display its contents at the end of each render loop. The functions currently provided are as such:

* + **DrawPoint**Draws a circle in 3D world space with radius specified. Note, radius is in world coordinates (e.g meters).
  + **DrawThickLine**Draws a line between A and B with colour and thickness specified. Note, thickness is in world coordinates (e.g meters).
  + **DrawHairLine**Draws a line a between A and B without thickness. (It will *always* be 1 pixel wide)
  + **DrawMatrix**

Draws rotation matrix at position/translation provided describing the X/Y/Z axes.

* + **DrawTriangle**Draws a triangle between vertices (A, B and C).
  + **DrawTextA**Draws text to a given world position, with text alignment specified.
  + **DrawTextF**Same as ‘DrawTextA’ except uses the same formatting (and optional parameters) as given by the ‘printf’ function. See printf for more information/formatting possibilities.
  + **AddStatusEntry**Adds the given text to a list of important data shown in the top left of the screen. Note these are cleared each render iteration so should be re-added each scene update or render call.
  + **Log**Adds the given text to the bottom left console text. This list persists over multiple frames, however has a limit of 20 entries in a FIFO system.

This is *your* framework now, so enjoy! ☺