Gravity Wells

Report

Submitted for MEng in
Computer Science

May 2017

by
Nick Ignat Smirnoff
Student Number: 201305389

Word Count: *1954*

Contents

1 Introduction

This document will focus on a single portfolio piece: “Gravity_Wells”. Focusing on the research, design and implementation of a distributed, physically-based modelling demonstrator, for a series of gravity wells. Simulating and arena of spheres and player controlled gravity wells through the use of graphics, physics and network components that run in concurrency with each other.

2 System Architecture

With reference to the class design (Appendix A: UML Diagram), “wWinMain” holds the entry point for the application, from which the application begins through the initialisation of two shared objects; configuration and content manager. The three core components of the application (Physics, Graphics & Network) are dependent on these two objects due to their nature of containing shared data and functionality relative to each component. Once the two shared objects are initiated the three main components can begin their cycle, on dedicated thread instances and operate separately from one another.

Configuration

The entire configuration object is initialised with default values, which are then overwritten based on the ‘client_config.cfg’ file should the values be in range.

```
cfg_window_width "1280"
cfg_window_height "720"
cfg_fullscreen "false"
cfg_vsync "false"
cfg_max_graphics_hz "60"
cfg_max_physics_hz "60"
cfg_max_networking_hz "60"
cfg_environment_size "100"
cfg_sphere_count "256"
cfg_sphere_weight_light "1"
cfg_sphere_weight_medium "2"
cfg_sphere_weight_heavy "4"
cfg_influence_region "10"
cfg_gravity_well_size "10"
cfg_gravity_well_power "10"
cfg_time_scale "1"
cfg_elasticity "0.5"
cfg_friction "0.9"
cfg_gravity "9.81"
cfg_ip_address "150.237.93.106"
cfg_port "9171"
```

Figure 1: *‘client_config.cfg’ contents.*

These user defined settings influence a number of characteristics the application operates under. The immediate use being the application window size and full-screen, an example of these settings are visualised in ‘Figure 1’ and are defined with appropriate token names.

Content Manager

The objects that make up the application are initiated and stored in this manager class; sphere list and a gravity well list. The objects data is shared and accessible across all three core components, each one using the data to carry out their functional responsibility:

- The Physics component; manipulates the data
- The Graphics component; renders the data
- The Network component; updates the relevant data

Physics Component

The Physics component is initialised, by picking up the two shared objects; from which the rules and objects can be defined. The component itself functions using an update method encapsulated in an infinite while loop within a 'Tick' method, which is managed through a timer class, allowing the control of how many updates can be called per second, and in turn controlling the behaviour of the object effected by physics.

```
std::thread thread_physics(&Physics::Tick, p_physics.get());
SetThreadAffinityMask(thread_physics.native_handle(), 0b100);
```

Figure 2: *Physics component set to run on core 3.*

'Figure 2' is a code snippet that sets the component in motion, by executing the 'Tick' method in a dedicated thread to run on core three. The update method is executed based on a user defined number and can be controlled both from within the application and the configuration file on launch. All the physics calculations and functionality is done within this update method, ultimately manipulating and updating the appropriate objects that are stored within the 'Content Manager'.

```
int div = (int)p_content_manager->cm_sphere_list_active.size() / 4;

std::thread collisions_1(&Physics::HandleCollisions, this, 0,       div);
std::thread collisions_2(&Physics::HandleCollisions, this, div,     div * 2);
std::thread collisions_3(&Physics::HandleCollisions, this, div * 2, div * 3);
std::thread collisions_4(&Physics::HandleCollisions, this, div * 3, div * 4);

SetThreadAffinityMask(collisions_1.native_handle(), 0b11111000);
SetThreadAffinityMask(collisions_2.native_handle(), 0b11111000);
SetThreadAffinityMask(collisions_3.native_handle(), 0b11111000);
SetThreadAffinityMask(collisions_4.native_handle(), 0b11111000);
```

Figure 3: *Physics component – Handle Collisions.*

The load of managing the physics across all active sphere objects is handled by splitting up the number of spheres across a number of threads and the possibility of five cores (cores 4, 5, 6, 7 and 8). 'Figure 3' demonstrates how the collision method is handled across four threads and each thread manages a quarter of the active sphere list.

Graphics Component

The Graphics component is initialised, by taking in the initialised window handle (HWND), the initial window width and height, and the two shared objects. The graphics component is responsible for handling the user input, and rendering the objects from the data accessible in the ‘Content Manager’. During initialisation, a camera object is created and is manipulated through appropriate user input. User input includes the simulation controls and window messages (resize, minimise, maximise, exit…etc.).

After initialisation the component can be executed to be executed in its own thread on a dedicated core (core 1). This is done through by setting the affinity of the current thread to function on the desired core, and is entered into a while loop that loops as long as the user does not push an exit/close request.

```
SetThreadAffinityMask(GetCurrentThread(), 0b1);
```

Figure 4: *Graphics component set to run on core 1.*

As long as there is no exit request, similarly to the ‘Physics’ component the loop calls a ‘Tick’ method which encapsulates the ‘Graphics’ update and render method, once again captured using a timer class allows control of the frequency of updates. This control being separate from the other two core components allows frequency control without having an effect on the behaviour of the other two. The update loop, handles all the user input and controls appropriately. Within the render loop the active objects are drawn on screen using ‘DirectXTK’ geometric primitives. The appropriate data of location and rotation and amount is all accessed and iterated through, from the content manager object; updates made to that are updated visually during render.

Network Component

The network component follows closely to the ‘Physics’ component, it is initialised by taking in the two shared objects, from which the initial settings can be defined and the functionality can be prepared.

Like the other two core components, the network update method is encapsulated in a ‘Tick’ method, and using a dedicated timer class allows control of the update frequency. The ‘Tick’ method is set to a dedicated thread and is carried out on core 2.

```
std::thread thread_networking(&Networking::Tick, n_networking.get());
SetThreadAffinityMask(thread_networking.native_handle(), 0b10);
```

Figure 5: *Network component set to run on the core 2.*

The update method sends message to other connected clients, the frequency of which can be controlled by the user. If the user exists the application, other peers maintain connection, should the user that exists be a host, the clients are able to break out of the main ‘Network’ thread and reinitiate through the same process as outlined in ‘Figure 5’ but as a new host if certain parameters are met; ultimately allowing continuous connectivity across active peers.

```
std::thread host_manage(&Networking::HostManageConnections, this);
SetThreadAffinityMask(host_manage.native_handle(), 0b10);
```

Figure 6: *Network component – Thread to listen to connections core 2.*

```
std::thread thread_process(&Networking::ProcessNewClient, this, client);
SetThreadAffinityMask(thread_process.native_handle(), 0b10);
```

Figure 6: *Network component – Thread to handle a new client connection on core 2.*

The 'Networking' component has a dedicated thread for listening to active connections and a thread to process new connections if there are available peer connections.

```
std::thread process_clients(&Networking::ProcessClients, this, clients);
SetThreadAffinityMask(process_clients.native_handle(), 0b10);
```

Figure 8: *Network component – Thread to handle active connections core 2.*

Active connections are managed on their own threads, 'Figure 8' is the process of an active connection executing; the thread listens for messages from that client, when more clients connect their messages are received on their own threads and are parsed on the same thread.

3 Simulation

The simulation is initialised from the configuration file, the user defined settings create the amount of sphere objects with desired parameters within the 'Content Manager' and then further used to define the behaviour of the physics that will act upon the sphere objects.

The simulation loop is main carried out in the 'Physics' component with manipulation to the rules being made in the 'Graphics' component. Euler's Integration is used to create the core functionality of the simulation loop; using the initialised objects, a force is calculated and used to calculate the velocity, this velocity is applied to result in a new position of the object, any dynamic collisions and responses are then applied should the conditions for them being met.

As briefly overviewed in the previous section, the 'Physics' component, specifically the frequency of the updates can be manipulated by the user; as the physics formulas used often integrate a delta time, low framerates greatly affect the Reponses and behaviour of the physics being applied. The user is able to completely pause the update method stopping all updates as well as slowing/speeding the process up through a time-scale value.

Spheres

The 'Physics' component update method starts of by initiating a background sphere-sphere collision check and updating the response appropriately. 'HandleCollisions' method has been briefly overviewed in the previous section as it takes advantage of multi-threading to functions; having a 'start' and 'end' values in a for loop a specified section of spheres can be checked against all the rest:

```
void Physics::HandleCollisions(int start, int end) {
	for (int i = start; i < end; i++) {
		for (int j = 0; j < cm_sphere_list_active.size(); j++) {
```

Figure 9: *HanndleCollisions – Check a defined sphere list section against all the spheres.*

Using this, a defined sections can be executed simultaneously, with each thread assigned a section. The check for collision is done by creating a distance vector, and a radius sum squared from the two spheres being checked, should the distance be smaller than the radius sum squared a collision has occurred. A force is calculated based of the absolute value of the penetration depth multiplied by 10 and that value stored to power of 2, and multiplied by 10 to complete the force formula. This force is then applied along the collision normal acquired from the positions of the two spheres. The process is complete by adding a slight energy dampen to the spheres.

The update method, iterates through each sphere object that is currently active (inside the area of influence), each sphere in this list first gets a gravity force applied to it:

```
ApplyForce(Vector3(0.0f, -( cm_gravity * s_mass ), 0.0f), p_delta_time);
```

Figure 10: *ApplyForce – Apply gravity force to a sphere.*

The 'ApplyForce' method is a member of each sphere and is called when an update to the current condition of the sphere is required as a response to a challenge or request. The result is a new velocity of the sphere object, by taking the force to be applied dividing by the sphere mass and multiplied by the delta time:

```
s_velocity.x += force.x / s_mass * dela_time;
s_velocity.y += force.y / s_mass * dela_time;
s_velocity.z += force.z / s_mass * dela_time;
```

Figure 11: *ApplyForce Method.*

From here environmental checks are carried out; sphere-floor, should the y-position of the sphere be equal to or lower than 0, the sphere has hit the floor and requires resulting reaction to bounce it up; depending on the elasticity parameter the ball is sent back in up in the y-axis based of the y-velocity of the sphere. The other environmental check is the spherical arena walls, sphere-wall; should the position of the sphere exceed that of the radius of the environment a force is calculated. This projection method calculates the difference between the spheres position and the spherical radius limit of the wall is the depth of penetration value, using this value the object is moved along the collision normal based of the affected velocity values, similar to that of a sphere-sphere collision.

After the environmental checks are carried out and all the necessary responses have updated the velocity, the sphere is checked against the gravity well to respond appropriately should it be required.

Friction is also applied to the horizontal axis (x-z-axis), as the sphere is on ground level and would have lost energy, and with each collision should the parameters match that of the user defined elasticity and friction the appropriate response is carried out to the updated velocity.

Finally the simulation loop is complete and the sphere is updated; a new position is calculated by taking the current position and adding the velocity forced multiplied by the delta time of the application:

```
s_position.x += s_velocity.x * dela_time;
s_position.y += s_velocity.y * dela_time;
s_position.z += s_velocity.z * dela_time;
```
Figure 11: *Position/Update Method.*

During this position update, the 'fake' rotation is calculated and updated for the 'Graphics' component to make a visualisation of the sphere; pre-update position is stored, the old and new position is used to work out the y-rotation. Next, using the absolute x-z-velocity values, multiplied by the delta time a speed of x-rotation can be determined:

```
s_rotation.y = atan2f(old_position.x - s_position.x, old_position.z - s_position.z);
s_rotation.x -= abs(s_velocity.x * dela_time) + abs(s_velocity.z * dela_time);
```
Figure 12: *'Fake' Rotation for a sphere.*

The updates are visualised through the 'Graphics' component when iterating through the active sphere list and creating a world matrix from 'YawPitchRoll' in the y-x-axis, and then translating the world matrix based of the sphere position.

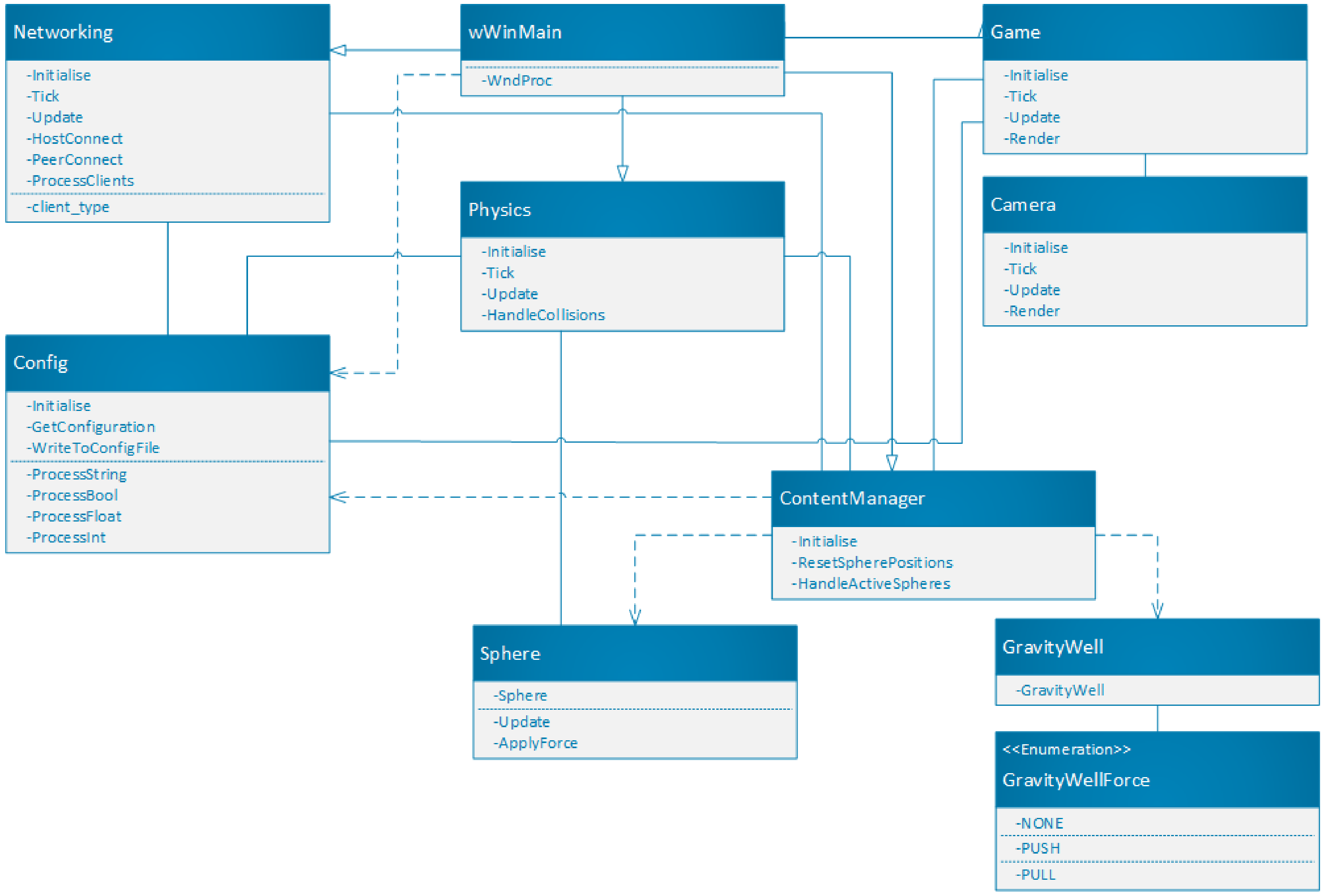
Gravity Wells

The 'Gravity Wells' are operated by the user, the position, the force and the forces power is all controlled by the user. Every iteration of a sphere object, the positions are compared to the position of the gravity well to work out if the sphere is inside the well, should it be inside the well the flag is turned to true. Sphere objects within the active sphere list that return true to being inside the gravity well are subject to an appropriate response based on the gravity well.

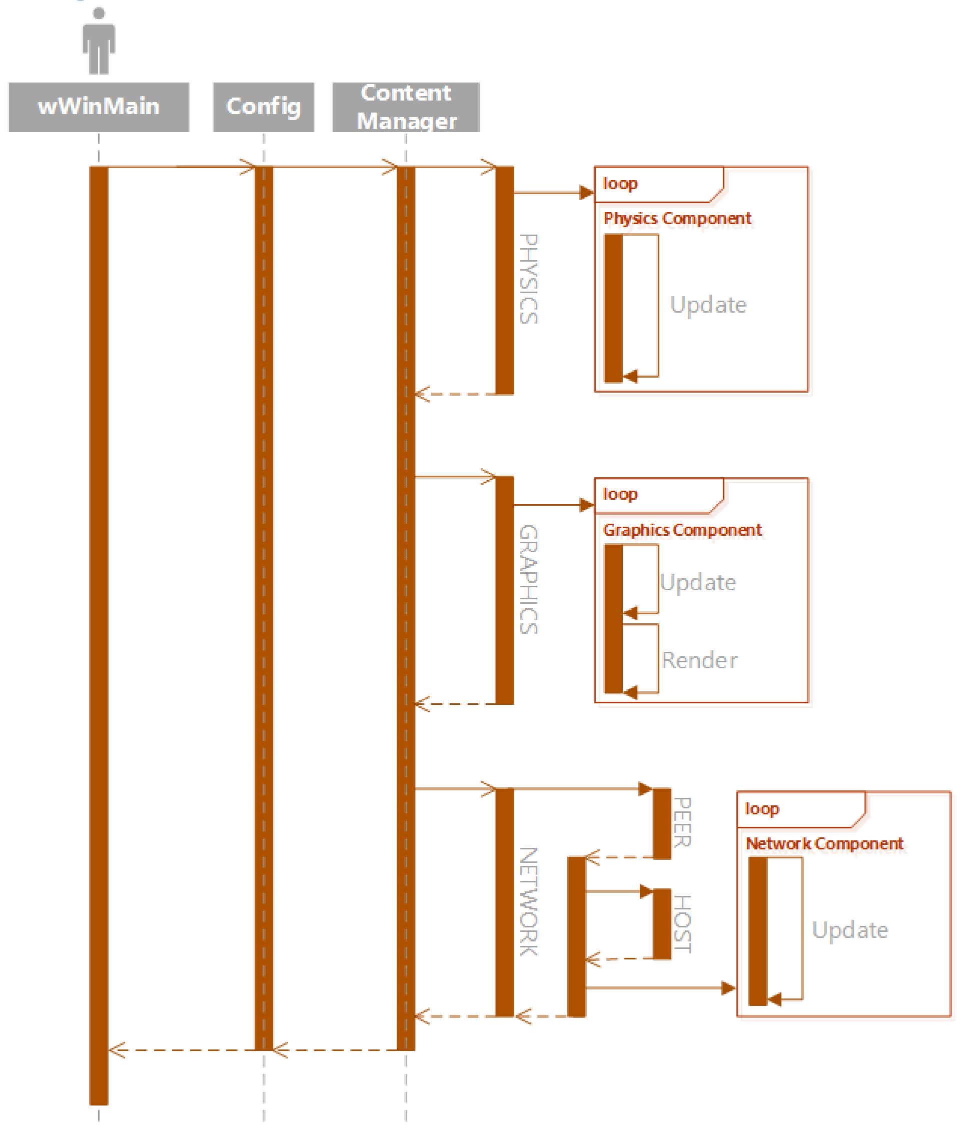
```
if (cm_sphere_list_active[i]->s_in_gravity_well == true) {
       switch (gw_force) {
       case GravityWellForce::NONE:
              break;

       case GravityWellForce::PULL:
              force_direction = Vector3(
                     gw_position - cm_sphere_list_active[i]->s_position);
              force_direction.Normalize();

              force = force_direction * gw_power;

              cm_sphere_list_active[i]->ApplyForce(force, p_delta_time);
              break;

       case GravityWellForce::PUSH:
              force_direction = Vector3(
                     cm_sphere_list_active[i]->s_position - gw_position);
              force_direction.Normalize();

              force = force_direction * gw_power;

              cm_sphere_list_active[i]->ApplyForce(force, p_delta_time);
              break;
       }
}
```
Figure 13: *Gravity Well – Check & Response on a sphere.*

Should the user be applying no force, the spheres are not affected. However, if the gravity well is active and the power of the force is greater than 0, the spheres within the well respond based on the direction with the force power; the force and the power is defined by the user, and is applied in the direction depending on the force itself. If the gravity well is applying a ‘PULL’ force, the direction is worked out by normalizing the position of the sphere taken away from the position of the gravity well, and vice versa for the ‘PUSH’ force.

Appendix A: UML Diagram



Appendix B: Sequence Diagram



References

cplusplus.com, 2017. *Thread.* [Online]
Available at: http://www.cplusplus.com/reference/thread/thread/
[Accessed 22 April 2017].

Dawkins, P., 2017. *Euler's Method.* [Online]
Available at: http://tutorial.math.lamar.edu/Classes/DE/EulersMethod.aspx
[Accessed 20 April 2017].

Microsoft, 2016. *DirectXTK.* [Online]
Available at: https://github.com/Microsoft/DirectXTK/wiki
[Accessed 15 April 2017].

stackoverflow, 2010. *C++ Winsock P2P.* [Online]
Available at: http://stackoverflow.com/questions/2843277/c-winsock-p2p
[Accessed 25 April 2017].