Simulation and Concurrency

## Architecture

The system architecture can be divided into several layers. I have produced diagrams representing the most significant interactions between classes related to specific activities. There are a number of utility and wrapper classes not present on the diagrams for clarity.

The following diagram represents the static state of physics objects within the application. Additionally the world class is interacted with by the application class for rendering and physics threads for updating the simulation.

The test collision method is overloaded in the base class for each derived object type to make use of the double dispatch design pattern so that specific collision detection routines may be called for specific pairs of object efficiently.



The following diagram represents the structure of the various threads responsible for tasks other than rendering. The diagram shows that the GameWorldThread is responsible for ownership of the network controller which includes managing update synchronisations, which is achieved by calling the ExchangeState method.

The network controller base class hides the implementation detail behind the session master and worker versions of the network controller, which are respectively responsible for accepting peers and making attempts to discover peers. Additional implementation detail shared by both controllers is handled by the ObjectExchange class which maintains a double buffer of update to receive and to send.



Alongside the classes present on these two diagrams a number of wrapper classes exist to abstract implementation detail behind graphics, networking and threading. Method calls to OpenGL, Winsock and Win32 threads were wrapped for clarity of implementation and to enable easier substitution with alternative should it be required at a later date. Many of the OpenGL wrappers were taken from the previous OpenGL snow-globe project and modified where necessary. A full list of wrapper classes is available in Appendix A.

## Threading

Threads are used throughout the project, all rendering and interaction with the GXBase window is performed on a single thread. Physics is primarily managed by a separate “Game World Thread”, which may also distribute the simulation work across a number of worker threads. Networking is performed on a separate thread which is created when creating/joining a session. To maintain synchronisation between the primary physics thread and the rendering thread the world class implements a triple buffering mechanism. This allows the rendering thread and the physics thread to run flat out, which is beneficial for benchmarking the simulation and rendering, whereas with a double-buffering mechanism the performance of the threads would be degraded by the least efficient thread. As the rendering thread performs no kind of interpolation it is not beneficial to render frames faster than they are updated, other than to measure performance, but it is beneficial for the stability and accuracy of simulation to update the physics simulation as frequently as possible.

The primary physics thread is responsible for synchronising with the network controller’s threads. The network controller maintains a double buffer of incoming and outgoing updates, which the primary physics thread consumes at the end of each simulation step. Because typically the networking thread spends a lot of time suspended while waiting on data, the double buffer mechanism was appropriate as it is often the case that there is not a complete network frame ready at the end of each physics cycle and so no action or synchronisation is required. The synchronisation between the primary physics thread and its workers is explained in more detail in the physics section of this document.

## Networking

There are two components to the network protocol, a UDP discovery protocol and a TCP protocol for maintaining the distributed physics simulation. UDP discovery was implemented to simplify the process of locating a session on the lab machines. In order to accept peers, a user must first press the ‘M’ key to create a session. When a session is created the application begins listening for broadcast packets and listening for connections on a TCP port. While the session creator doesn’t have a connected client it replies to broadcast messages with its TCP port. The connecting client, in join mode, receives this port number and initiates a new TCP connection.

Upon initiating a TCP connection the connecting peer is sent a block of initialisation data which includes the type and states of the objects in the session creator’s simulation. After initialisation each peer sends state updates to the other peer for objects it owns. Additionally each client may request to take ownership of an object. Ownership requests are made for objects such that objects on the left half of the world belong to the first peer and objects on the right half belong to the second peer. Additionally the peers will request ownership of objects which have the user’s spring attached. If an object is requested that the other peer already has the spring attached to it may refuse to transfer ownership as there is a confirmation message that must be received before a client takes ownership of an object.

All calls to Winsock have been wrapped into classes that are similar in structure to those found in .NET. Throughout the entirety of the network protocol implementation (found in NetworkController.h, .cpp) not a single call to Winsock is made directly, which allows for additional abstraction to be separate from protocol implementation. For example the Message class implements serialisation and deserialization to a binary format and TcpSocket’s receive handles capturing entire messages from the stream without returning partial messages.

To provide fault tolerance the network controller retains the last updated location for objects it does not own. When a complete frame does not arrive within a timeout period the connection is terminated and the positions of objects are restored to their last known position. To maintain a relatively smooth simulation the states of objects are integrated by both connected peers. However the collision response is only calculated for objects that are owned by a peer. Provided that a peer knows all of the forces acting on an object objects maintain a smooth motion. The simulation becomes noticeably jumpy when the forces are not known by another client which tends to be in two scenarios. As creation of the user controlled spring is not networked, the object will not move as smoothly as objects only under the force of gravity. Additionally when an object is at rest and there is a brief network pause or a timeout objects may begin to fall through the floor and then re-appear as collision response is not performed for objects that are not owned by the peer.

## Physics

Physics simulation is broken down into multiple stages, each of which may be distributed across any number of worker threads. The stages in sequence are, integration, broad-phase collision detection, narrow-phase collision detection and collision response. Upon completing each stage the primary physics thread waits for all workers to complete the stage before starting them on the next stage. For optimal performance on a quad-core machine the number of worker threads is currently hardcoded at two (including the primary thread) as the networking and rendering require an additional two threads. It should be possible to scale this to any number of threads and the performance should scale with it when there are a sufficiently large number of objects. Unfortunately I have not had access to any hardware with greater than a quad-core to test how well it scales beyond two worker threads.

### Integration

Runge-Kutta fourth order integration is used to update the state of each object. During integration the global forces such as gravity and drag are applied and the forces of all local constraints, such as spring constraints, are calculated. The integration phase is divided across the available physics worker threads such that each thread is responsible for an equal share of all objects.

### Broad-phase Collision Detection

Following integration is the broad-phase stage. To facilitate broad-phase collision detection the world is divided into a 2d grid of buckets. Each bucket contains a pointer to all objects for which the centre of mass belongs within that grid-cell. This allows the narrow-phase collision detection to consider only possible collisions between objects within the same or neighbouring grid-cells. Which vastly cuts down the number of possible collisions to test provided the buckets are small. Because only collisions with neighbouring cells are considered this places a constraint on the spatial hash grid that each cell is at least as large as the bounding box of every object. The implementation could be expanded to allow for much larger objects at the cost of complexity.

When this task is distributed across a number of worker threads each thread visits each object (but must not modify them) and maintains the cells in a vertical strip allocated to that thread. For example if there are two worker threads, the first thread would be responsible for maintaining the lists in the left 50% of the cells and the second thread the right 50%. A vertical arrangement of cells for each thread was chosen because it provides an even distribution of objects across both threads as gravity is acting in the y-axis. A more general solution might allocate each thread cells in a checkerboard pattern for example.

### Narrow-phase Collision Detection

After the broad-phase stage, a list of contacts for each object is generated by testing it against each object in the same grid cell and neighbouring objects. Each type of object implements its own collision detection routines against other types of object, for example boxes detect collisions against other boxes using an AABB intersection algorithm. Double –dispatch is used to determine which collision detection method to invoke for a pair of generic physics object.

Narrow-phase is distributed across multiple threads by allocating a section of the grid generated in the broad-phase stage to each thread in a similar fashion to that described in the broad-phase stage, with the only difference being that the grid should now be considered read-only and the objects contact list should be considered read only to threads other than the one owned by it. This method of distribution was chosen because it enables an optimisation in that when a pair of objects fall within cells owned by same thread the collision detection would only need to performed once per collision as the contact data may be reversed and applied to the other object in the pair.

### Collision Response

After collecting the contact data for each object the objects are distributed across the worker threads such that each thread receives an even share of objects. For each object the collision response for each contact point is calculated. Collision responses are treated as impulse forces and are repositioned by separation distance. The collision response assumes particle motion because sufficient contact data for rigid-body motion is not generated. To prevent issues with stacking the contacts are sorted by contact normal projected onto the gravity vector, which allows them to come to a rest without sinking as objects tend towards the response away from gravity.

## Appendix A – Table of Wrapper Classes

The following wrapper classes were developed to simplify implementation, provide additional abstraction over libraries and enable ease of porting to equivalent libraries at a later date should it be required.

|  |  |
| --- | --- |
| **Graphics Class** | **Description** |
| Renderer | Provides a thin wrapper around the OpenGL state. Provides a non-deprecated implementation of lighting, clipping plane and model, view, projection matrices. Provides OpenGL extensions to graphics objects |
| Shader, VertexShader, FragmentShader, Geometry Shader | Provides a wrapper around individual GLSL vertex, fragment and geometry shaders |
| ShaderProgram | Provides a wrapper around GLSL shader programs |
| Uniform | Provides a wrapper around the location of a uniform variable so that it may be cached for updating the uniform later |
| VertexBinding | Provides a wrapper around the vertex attribute state, utilises OpenGL’s Vertex Array Objects to provide a mechanism for efficiently re-binding vertex buffers |
| VertexBuffer | Provides a wrapper around OpenGL’s Vertex Buffer Objects which store vertex and index data in graphics memory |

|  |  |
| --- | --- |
| **Networking Class** | **Description** |
| Address | Provides a wrapper around the network address struct that handles conversion to and from string formats |
| Message | Provides a mechanism for serialising data to a fixed maximum size byte array. Additionally provides a deserialization mechanism for retrieving data from a byte array received over the network. |
| TcpListener | Provides a wrapper around a TCP socket that listens for incoming client connections |
| TcpSocket | Provides a wrapper around a TCP socket that is created to establish a connection to a TcpListener or is received as part of accepting a client through a TcpListener |
| UdpSocket | Provides a wrapper around a UDP socket that may function in ordinary or broadcast modes |

|  |  |
| --- | --- |
| **Threading Class** | **Description** |
| Event | Provides a wrapper around a synchronisation event |
| Mutex | Provides a wrapper around a mutex that should preferably be acquired and released through the use of a ScopedLock object |
| ScopedLock | Provides a mechanism for acquiring and releasing a lock that subscribes to the RAII design pattern |
| Thread | Provides a wrapper around a thread. Thread should be inherited from and the ThreadMain method should be overridden with desired thread functionality |