Multithreading the update and draw loops

We do not mind if the draw loop lags slightly (1-2 frames) ... the update loop should always be faster than the draw loop ... if not the draw loop will simply redraw the same image (or even better ... do nothing). The big worry is that this will introduce an ‘input lag’ – however this is shelved for now.

The plan is to have some kind of buffer structure that is possibly static. The general problem of threading is that objects need to read from and write to fields, but there is no guarantee that all data is not in between writes to several values before it reads /writes something. Part of the solution to this is to force the fields of each thread to be independant to ensure no cross talk. This however creates a problem of cheap communication between the 2 threads. This is where a buffer comes into play. One thread generally consumes the data of another thread and then perhaps publishes data for another to consume. The buffer essentially creates several frames of data and only lets a thread consume a frame if it is complete. Likewise the producer thread can only start producing data if there is an available buffer, hence the need for 2 buffers. However it could be possible to have 3 buffers to allow the producer to perform more work whilst the consumer deals with the data. This also introduced the possibility of combining buffer frames in some cases where addition can be used – this may introduce the idea of quad buffering if the structures contain only differential information. Obviously if they only need to contain absolute data then triple buffering will suffice as each frame replace the previous – the detail of the application of the structure can be in the object that the structure it will be applied to.

Because of this the update loop will always operate on the central objects and the draw loop will operate upon a list of structures generated by the update loop in addition to the central objects. These aspects of the objects are separated logically by interfaces. This means that both sides of the object may contain ‘position’. By this definition, everything that is not to do with drawing is part of the update loop and so is not simply restricted to the physics update in here. This separation effectively creates a buffer of information in each object. Later there may be more interfaces tret like this EG the sound loop. The structures are what is used to transmit updates to these other regions in the objects when these separate regions are to be up onto separate threads. This allows each object to control its implementations of the functionality these interfaces outline.

The problem of how to reduce the number of objects the draw loop has to iterate through will be dealt with at a later date. The idea of a game space unit is what will probably be used ultimately to solve this. The current thinking is to subdivide game space units into 8 – 27 could be considered as it has a center and so could have a neater fit. The thing is to remember that in a client server game the player would not have to do the collision calcs etc as that would be done server side- ie the problem there is largely a drawing problem and prediction (which could but nto necessarily involve collisions). IE when it becomes a problem things might of changed a lot.

The structures are everything necessary for the draw loop to update the drawing information about the object. The objects to be drawn need to contain a reference to a draw method which can be achieved through an interface (IDrawable). This should also define a method to allow a drawing structure to be applied. This allows each object to have responsibility for its own rendering. For instance the DrawingStructure currently contains the models position and rotation as vector3 and quaterion as values.

There is also a buffer class that contains at least 2 lists of each buffer structure type. Its

The draw loop in the DrawingLoopComponent will contain a list of DrawingStructures, each containing updates to the drawing information. It also needs the Camera that the objects to be drawn require. It will perform the logic to determine which objects are in view (if the camera moves than all objects need to be tested for FrustrumVisibility again, if not only changes matter in comparison to existing FrustrumVisibility state).

This list will get updates from when the update loop adds to a queue of changed objects, new objects and objects to be destroyed.

Both of the threads get a reference to the queues passed to them on creation. The only thing that need to be ‘multi-threaded’ here is the queue ... but as one thread will add and the other remove there isn’t a problem here either.

Multithreading inside these threads can be considered – however i am cautious as context switching is a \*bad\* thing. If you exceed an optimum amount (for the program – load will vary with time) the performance plummets. With no way to precalculate this varying edge things are very difficult.

The main use for it will be File IO / networking etc anything we need asynchronously and is order independant.

# Multithreading the Game Engine:

## ThreadPool:

.Net has a threadpool class which offers a managed threadpool that expands and contracts the number of threads based on cpu utilisation.

So, we use this to pull our threads from. This is also a singleton class accessed via static methods and so it can be utilised anywhere in our code. The key question will be identifying where.

## Thread Independance:

If we have 2 threads then the thing we must try to avoid at all costs is invoking methods on threads other than the current thread. This is expensive and so requires designing around. This means that the threads must contain a reference to the objects (Relevant objects will implement the appropriate interface) that it requires to do its work.

## Thread Communication:

The problem of cross thread communication can be handled by passing a reference to a buffer of structures into a thread. This buffer will contain any information that it requires up update. This implies that the buffers structures are then dependant on the type of interface it updates.

## BufferStructure:

A buffer structures fields will be the unique object ID, and the fields required to update the implementor of the interface.

As interfaces get inherited and complexity increases structures cannot mirror as they cannot be inherited. This forces us to force the layout of the structures to allow us to effectively override its data format. If we know the size of the datatype being stored then we can specify the next’s offset. By overlapping offsets we can construct multiple uses for the same structure, we could use a enum field to identify the particular use.

## Program Flow

By constructing threads and buffers for them to pass information down the line of threads and have freedom on the number we use. This will allow the first thread to start work on the next work unit. Currently 2 threads have been identified, the update and draw. These can have a high degree of separation but we recognise that the update might want to be split some more. Currently Draw and Update are in the GameSpaceUnit but this causes problems.

However there is a choice in flow type, or to put it another way, how many humps do we want in the caterpillar? Also or caterpillar will have its head joined to its tail because as it finishes it will start again. IE Do we really want to be starting another update before the last has completed?

(Right now we have a draw loop, but as a server this is not the case, it is a pure update loop + network handling)

Isn’t the point of the threadpool that you just set it up and it works it all out? ...YAGNI

## Other Game Components

Assuming we are constructing a single player game. We need to be handling input, handling the user interface for the game, Managing a game state enum(eg exit,paused,in menu),be able to drop back to a menu screen. This means that GameSpaceUnit is just one part of a larger system.

### UserInterface:

The user interface isn’t so bad, first off, it is currently 2D and so is obviously separate in that it uses a different draw call (spritebatch), also when it is receiving input the other component presumably isn’t. Essentially it can be done by catching the events from the event manager in a UIControlManager which will establish wether a control is active. If not then the input is passed to the next object that requires it. Currently the only other object is the GameEngine.

### GameEngine

The game engine is the component that holds the game objects in a GameSpaceUnit. This class provides the update and draw functionality for all objects in the 3D portion of the game. It is the 3D portion that will be discussed unless stated otherwise. Right now, we only have one GameSpaceUnit. As such this class is redundant. It will be resurrected when we have a need for multiple GSU’s.

#### Update

We could carry out types of operation seperatley, however this requires either iterating through collections of objects multiple times in a parallel fashion (if multi threaded completely) – functional threading - or iterating through the collection in a parallel mannor and each object performing multiple operations depending on its inheritance structure – data structure threading. The decision will be balanced by the time to iterate through the variable numbered and sized collections vs the cost of getting and returning the number of threads equal to the number of updatable objects. The latter isn’t strictly true, essentially if there is a thread available the code can spawn a thread for a number of work units and then reevaluate again upon completion whilst performing work units itself. See below for why i choose the second option – please comment if you want, its just my reasoning, I’m not proud ;).

There is a need to be able to run certain methods before and after the multithreaded part of the update simply because they need to be guaranteed to be completed before or ran after. If my assumptions about gravity are wrong for instance, then it can move into the ‘prior to threads’ category.

##### Collisions

No matter which way we choose we still have to establish which objects were updated. However there may be different kinds of update. Not all update may result in a collision and so we are interested in any updates of objects that implement IPhysicsObject as this is the Interface that very weakly implies ability to collide. These can be determined by examining the object referenced in the displacement structures.

##### Gravity

There is a requirement that a list of objects that emit a gravitational field be kept to minimise the number of iterations required for each object effected by them. This is a hideous n(mx2) problem that will require babysitting, but is essential – I am having ideas about using HLSL/CUDA for calculations if we can get data back out. RICK: This would allow massively parallel computation if you fancy a challenge – however it will only be useful for a server app as we do not want to limit our uses to be purley nvidia.. The gravity calculation can be done in each object unless some genius manages to code me the above. In which case we add a buffer and have each frames gravity lagging slightly but pull data out of the draw into this for the update to munch. This feels like a circular dependence unless you make the gravity update use the last available values (at startup = 0) if there are none from the uber gravity calcs and make it use a buffer – ie treat it as a late thread. The assumption is that the costs of sending data are high but calculation tiny. This will result in a large reduction of update time and possibly the gravity update taking longer than the main update. Also the changes in gravity will be gradual meaning that having totally up to date data is not necessarily essentially. Howver once the number of objects int he game increases the gravity computations are going to become interesting.

##### Propulsion

Some objects have the ability to propel themselves and require these updates. All objects that can be propelled have thrusts that have an orientation. Every update these need to be calculated and published

##### ExternalControl

Players and AI need to be able to carry out methods upon whatever objects they can have to use. At the most physical level they need to be able to (wrt time) apply a rotation vector and a translation vector/enum/bool that can be translated into Propulsion commands.

##### Argument for a single iteration loop

Personally I think the limitation is the knowledge that running more than 3 threads of computational work is not going to work if there are not 3 cores available (ok, you have processor specific architecture but sadly this is a lowest common denominator problem – until cores on all platforms run multiple simultaneous threads we have to assume threads = processors, intel users will get more from this i imagine due to hyperthreading). As such by using the first method you know you are going to have to traverse a decent proportion of the game objects multiple times. By using the second method you get it all in one variable whack.

The design choice is for the single iteration update loop as it should be more flexible (moreover I know that .net4.0 has a for each parallel function that we can utilise). Alternativley we can use lambda expressions to construct a functional equivalent ourselves. If we write in as functional way as possible then multithreading in the future will be easy.

There always needs to be capacity however for some methods to be guaranteed to be run first. Gravity appeared to be one of these, however it is not because the acceleration due to gravity isn’t really useful for any other objects apart from its owner. Please correct me if wrong. However this highlights the need for certain operations to be carried out before and after the threaded part of the update. This poses the problem of figuring out which objects have updated to avoid iterating the whole set again

This creates a requirement to have any displacement structures that the objects need to be generated prior to the collision resolution