CGIT Final Report - Flocking

The flocking program uses many of the same techniques covered in labs with regards to QT user interface, VAO object usage, CMAKE structure etc. For brevity, only the techniques and approaches specific to the flocking project are mentioned here. Source code should be adequately commented to explain any part of the project, however.

Aside from extraneous code used for the user interface and scene management, the core flocking algorithm itself exists entirely within one flock class. In my initial class diagrams, I had opted to create an “Agent” class, inherited by both “Boid” and “Predator”. In order to reduce the scope of this project, I elected to discard the predator class and folded the “Boid” class into “Agent”. However, this method stored a lot of unnecessary information about each agent that would not be required for many functions. In addition, it was unlikely that tuning parameters for individual agents would be feasible, given the number of them. For these reasons, instead of holding an array of large agent structures, the code was refactored so that a single “Flock” class would hold a vector of pointers to a simple Agent structure holding both position and velocity. Parameters shared among agents (for example size) were then stored by the class outside of any data structure. Keeping these separate from one another helped ensure that only necessary data was passed around the program, improving performance. Upon reflection, separating this flock class into different classes would have had little utility. All attributes and methods of the flock class are private, other than those used by other classes to access the flock (attribute setters used by QT, constructor, update and render methods called from an NGL scene). This allowed for encapsulation, preventing external classes from interfering with the internal workings of the Flock class.

Before delving into more specifics of the code it will be useful to explore the core flocking algorithm, in order to give context to everything else. This algorithm is based heavily on that of Craig Reynolds (Reynolds, 1987), as discussed in the background research essay. Upon each frame update, each agent has a new acceleration vector calculated. This is then scaled by the “m_maxSteering” parameter which controls the amount of steering force (a low value will result in slow smoother changes in direction and vice versa). The velocity of the boid is then modified using the equation $\Delta v = a \Delta t$ (change in velocity = acceleration * change in time). The new velocity is then scaled using “m_maxSpeed” which controls the overall movement speed for the agents. Using the equation $\Delta x = v \Delta t$ (displacement = velocity * change in time) the new agent position is then calculated. In both of these cases Δt is a time step calculated within the NGL scene (time taken for the flock to update). Using this time step means that if the program slows down due to large numbers of agents, they will still move in real time.

The flocking acceleration is a scaled sum of 3 different acceleration vectors (4 including collision avoidance). These vectors correlate to Reynolds 3 rules of flocking: separation, cohesion and alignment (or collision avoidance, flock centering and velocity matching). Each is scaled by a parameter controlling their weight, allowing for the different rules to be given priority over one another as the user sees fit.

The separate rule ensures that agents do not get too close to one another. A desired separation distance is specified by the user. For each agent, for any agents that lie within this distance the normalised position vector difference between the two is found and divided by the distance between the agents. These differences are then averaged across all agents that are too close. Subtracting the agent’s current velocity from this average gives the steering force needed to move away from any nearby neighbouring agents.

The cohesion rule is almost the inverse of the separate rule, ensuring that agents try to stay close to one another. A user controlled parameter (m_neighbourhoodRadius) gives the viewing radius of each agent. The position of every agent within this radius is averaged and the agent is steered towards this average position via a simple seek method. This seek method first subtracts the current agent position from a target position, then subtracts its current velocity from this. Here, unlike separate, the exact distance between neighbours is not used, instead the square distance is just as useful and is faster to calculate.

The third rule, alignment, is very similar to the previous but attempts to match velocity with other agents, not position. The velocity of all agents within the neighbourhood radius is averaged. This is then normalized, and the current velocity is taken away to give the final steer force. Removing the normalization here was trailed to improve speeds, but the effects were far worse.

Aside from collisions with each other (which are handled by the separate rule) agents can collide with either the edges of the cube in which they flock (the "scene cube"), or with several cube shaped obstacles which can be added to the scene via the QT user interface. Collisions for both cases are handled differently. Collisions with the edge of the scene cube are checked for at the end of the update method, after a new position and velocity have been assigned to an agent. The user can control the agent's behaviour upon wall collision via the UI, the two options being bounce or teleport. If set to teleport, upon reaching a boundary of the scene cube, the position of the agent in the colliding axis is simply negated. For example, if hitting a wall at y=4.5, the agents y position will be set to -4.5 and they will fly from the other side of the scene cube with the same velocity as previous. If agents are set to bounce off the walls then a value of 3 is simply added (or subtracted) from their velocity in the colliding direction, making them fly in the opposite direction towards open space. Both methods worked well until the time step mentioned earlier was added. If the delta value (time step) was high (due to slowdown between frames) then agents could drift outside of the scene cube boundaries during the delay. To prevent this, upon detecting that an agent position is outside of the scene cube, the position is modified to place the agent at the edge of the cube. This ensures that even during drift caused by frame rate slowdown, the agents will never exit the scene cube. This guarantee of position bounds was especially necessary to prevent segmentation faults when implementing spatial partitioning (discussed later).

Collisions with obstacles are handled in a different manner, a series of 4 cubes is hardcoded in to act as obstacles which can be toggled on and off. Each cube has an associated bounding sphere which encompasses the cube. A 4th "collision avoidance" rule was added to return a steering force which prompts the agent to avoid obstacles. This avoid force is not scaled down with a user parameter like the others, giving it a higher priority when they are summed together. Collisions are calculated using ray sphere intersection due to its high performance speeds. A "see ahead" ray is generated, scaled to be of length 3 (many values were tested but 3 gave the best balance of pre-avoidance and not avoiding too soon). This ray is then scaled by the delta time step to ensure that collisions are avoided even during the drifting periods that occur during slowdown. For each obstacle, a Boolean returns whether the ray will hit one of the obstacles. The ray sphere intersection method here is one commonly used in ray tracing. The ray start and direction are passed in, as well as the sphere centre position and radius. A sphere can be represented as *||P – C||^2 = r^2*, where *P* is a point on the sphere, *C* is the spheres centre and *r* is the radius. This is equivalent to *(P-C) dot (P-C) = r^2*. A point on the ray can be expressed as *p(t) = A + Bt* where *t* is a time step, *A* is the ray origin and *B* is the direction. Substituting *p(t)* for point *P* in the sphere equation we get *(A + tB - C) dot (A + tB - C) = r^2*. Once expanded and rearranged this forms the following quadratic equation: *t^2 B dot B + 2t B dot (A-C) + (A – C) dot (A – C) -r ^2 = 0.*

When written into the quadratic formula, the discriminant is *b^2 – 4ac*, where *a = B dot B, b = 2(B dot (A – C)) and c = (A-C) dot (A-C) – r^2.* This discriminant of the quadratic equation lies inside a square root, so the equation is only solvable if *b^2 – 4ac* is >= 0. As such the code simply checks if this inequality is true and returns the result. If the result is false, no intersection has occurred (and vice versa). The closest obstacle for which intersection occurs is steered away from by subtracting the centre position of the obstacle from the ahead ray vector. This steers the agent towards the edge of the bounding sphere, avoiding the cube obstacle within. It results in some visually pleasing behaviour in which flocks will split apart and reform to avoid the obstacles.

Code speed was critical for rendering as many agents on screen at once as possible, as such a lot of time was spent refining the code to make it more performant. As well as a few already mentioned, various techniques were implemented to cut computation time. Wherever possible expensive square root and normalisation operations have been removed. The shader is purposefully minimal, simply taking in a position and colour as well as an MVP transformation matrix used to apply camera and agent transformations. This colour is then passed to the fragment shader to be assigned per fragment. With flat colour, agents were initially quite had to pick apart from each other, so adding some phong shading within the shader was considered. However even though shader code is generally fast to execute, doing lighting calculations for large numbers of agents could have still resulted in some slowdown. Instead a more lightweight solution was implemented, simply colouring the bottom half of the agent mesh a darker shade of green to the top. This cheat looked comparable to real shading and offered much better contract between agents with no additional cost.

The method of spatial partitioning (discussed in the prior background research essay) was implemented in order to improve computation speeds. In short, the scene cube is cut up into a series of partitions, and agents only consider other agents within their own partition and all immediately surrounding ones, with partition membership being updated as agents move through the scene. This radically reduces the number of agents being passed into the flocking algorithm and improves speeds greatly (provided that agents are spread around the scene cube). The number of partitions is driven by the neighbourhood viewing radius set by the user. Any time this radius is changed, spatial partition membership is recalculated. Partitions are elements of a 3d 20 * 20 * 20 array, which hold pointers to the agents within them. The minimum number of voxels is 1 (a single 9 by 9 cube encapsulating the entire scene cube), the maximum number of voxels is 8,000 (where each is a 0.45 by 0.45 cube). The voxel index used for an agent is calculated as follows (shown here for the x index): *binX = int(agent.pos.m_x * 1 / neighborhoodradius) + offset.* Where *offset = (int)(1 / _radius * 4.5). (With 4.5 being half the height of the scene cube).* This method ensures that bins are assigned such that only the current bin and its immediate neighbours need to be queried, to view all agents within the viewing radius. This dynamic allocation of bin sizes results in some small slowdown upon changing the viewing radius, but is much faster than concatenating lots of small bins together each update. This method means that the smaller the viewing radius, the faster the flock will run at the cost of agents being able to observe a smaller number of neighbours. It is very effective when large numbers of agents (1,000 +) are employed.

Initially, the vertices held in the agent VAO were recalculated and re-assigned upon each update, with changes to agent size, orientation and position being directly applied to these vertices. This was slow so instead of modifying the vertices each time, transformations were applied to the model view project (MVP) matrix fed to the shader (MVP = MVP * translate * scale * y_rotate * x_rotate).

Speeds were still sub optimal even after implementing this, however. After profiling the code using the QT creator IDE, it was discovered that most time was being spent assigning and creating new vector values. To combat this all Vec3 values are stored as flock class attributes and reassigned, rather than being created repeatedly on each function call. Instead of indexing into arrays, pointers to agent data structures are used by the flocking functions. This allows direct access to their speed and position without indexing into an array.

With all of these fixes implemented, the flock can comfortably run with 3,000 agents given a small enough neighbourhood radius. More than this will still cause slowdown however. Future work could be done to implement threading which would improve speeds even further.

Given the nature of a flocking algorithm, very low-level testing of behaviour is not feasible (e.g. At this point in time, where should this agent be?). Instead, the best way to test the functionality of the flocking algorithm itself was by eye, with it being quickly apparent when flocking was not occurring as expected. With that being said, some lightweight unit tests were useful to ensure that components were not broken during development. Tests were only written for the flocking class, as other code is heavily adapted from other sources and was already tested. In many cases it is argued that only the public interface should be tested. However, here it did not make sense for the majority of flocking methods to be made public, nor did it make sense to add an arbitrary public interface just for testing purposes. Despite this, it would be unwise to only test the public functionality of the flock, so friend functions are used as a solution. Adding the tests as friend functions to the flock class allowed the tests access to private attributes and methods. These tests were written in accordance with test driven development, a failing test was written, then enough code to satisfy the test and repeat. This was done to test the constructor and update methods as well as the various QT parameter setters. Due to the code structure, drilling down further than this was not necessary, as a break in any of the components within the code would affect one of these three areas, resulting in a failing test.

Refs:

Reynolds, C., 1987. Flocks, herds and schools: A distributed behavioral model. ACM SIGGRAPH Computer Graphics, 21(4), pp.25-34.