15-462 Project 5: Flocking and Animation

Release Date: Monday, April 04, 2011
Due Date: Monday, April 25, 2011, 23:59:59

Starter Code: http://www.cs.cmu.edu/afs/cs/academic/class/15462-s11/www/project/p5.tar.gz

Useful references:
OpenGL Reference Pages: http://www.opengl.org/sdk/docs/man/
Boids Flocking: http://www.red3d.com/cwr/boids/
Steering Behaviors: http://www.red3d.com/cwr/steer/
SDL Documentation/Tutorials: http://www.libsdl.org/

1 Overview

In this lab, you will learn about flocking behavior and animation. The project consists of writing an interactive application. The application will focus on a flocking algorithm implemented with a particle system. Rendering will be done with OpenGL.

After a (not-so) brief interlude with raytracing, we will be returning to OpenGL, as this will be a real-time, interactive application. As always, the OpenGL Programming Guide is useful. You will also find the online documentation for SDL useful for implementing the “interactive” part of the application.

The project is intended to allow exploration. There are no hard requirements for the actual behavior of the application; the only requirement is that it utilize a flocking algorithm as described in this writeup. So feel free to make this final project as interesting as possible, from both technical and creative perspectives. In other words, we’d like you to make something awesome.

2 Description

The main focus of this project is the flocking algorithm. You will implement simple flocking behavior as described by the popular “Boids” flocking model.

Particle simulation is one of the simplest types of simulation, but can be extended to animate a wide variety of phenomena such as rigid bodies, fluids, hair, cloth, etc. It has practical applications in all sorts of simulation software, cinematic special effects, and video games.
You will be using a particle system to implement a flocking algorithm. The particle system will track the position of velocity of the flock members, or “boids,” and the flocking algorithm chooses appropriate forces to apply to the particles to provide flocking behavior.

3 Submission Process and Handin Instructions

Failure to follow submission instructions will negatively impact your grade.

1. Your handin directory may be found at
   /afs/cs.cmu.edu/academic/class/15462-s11-users/andrewid/p5/. All your files should be placed here. Please make sure you have a directory and are able to write to it well before the deadline; we are not responsible if you wait until 10 minutes before the deadline and run into trouble. Also, remember that you must run aklog cs.cmu.edu every time you login in order to read from/write to your submission directory.

2. You should submit all files needed to build your project, as well as any textures, models, shaders, or screenshots that you used or created. Your deliverables include:
   - src/ folder with all .cpp and .hpp files.
   - Makefile and all *.mk files
   - writeup.txt
   - Any models/textures/shaders needed to run your code.

3. Please do not include:
   - The bin/ folder or any .o or .d files.
   - Executable files
   - Any other binary or intermediate files generated in the build process.

Run make clean before submitting. If you were using Visual Studio, be sure to clean the solution before submitting.

4. Do not add levels of indirection when submitting. For example, your makefile should be at .../andrewid/p5/Makefile, not ...

5. We will enter your handin directory, and run make clean & make, and it should build correctly. The code must compile and run on the GHC cluster machines. Be sure to check to make sure you submit all files and that it builds correctly.

6. The submission folder will be locked at the deadline. There are separate folders for late handins, one for each day. For example, if using one late day, submit to .../andrewid/p5-late1/. These will be locked in turn on each subsequent late day.
4 Required Tasks

A very general overview of the implementation requirements is as follows. Refer to subsequent sections of the handout for more details.

**Input:** We provide you with an application and empty functions to implement your program. We also provide you a sample executable of a flocking simulation.

**Output:** You must fill in the functions provided with your implementation of a particle-based flocking simulation. Furthermore, you must make it interactive by using either mouse or keyboard input that allows the program to demonstrate correct flocking behavior as specified below.

Correct flocking behavior will be defined as following the four rules laid out in the Boids model for flocking. Specifically, your flock must be able to demonstrate these four rules and additionally, handle orientation realistically as described above.

An example of what we would be looking for is as follows: a simple scene where a flock of particles (rendered as models) follows the mouse. Additionally, while there are no restrictions on how you choose to structure and implement your code for this assignment, you are not permitted to use libraries or flocking code found externally for this project.
Requirements:

- Implement a particle simulation with a stable integration scheme.
- Implement a flocking simulation using the particle simulation and the four flocking forces: separation, alignment, cohesion, and goal-seeking.
- Render the boids (i.e., flocking agents) using meshes or another nontrivial method.
- Orient the boids as specified in this writeup to simulate banking.
- Create an interactive application or game that uses the flocking algorithm.
- Submit a few screen shots of your program’s renderings.
- Fill out writeup.txt with details on your implementation.
- Use good code style and document well. We will read your code.

At a minimum, you must modify game/main.cpp and writeup.txt, though you may modify or add additional source files. writeup.txt should contain a description of your implementation, along with any information about your submission of which the graders should be aware. Provide details on which methods and algorithms you used for the various portions of the lab. Essentially, if you think the grader needs to know about it to understand your code, you should put it in this file. You should also note which source files you edited and any additional ones you have added.

Examples of things to put in writeup.txt:

- Mention parts of the requirements that you did not implement and why.
- Describe any complicated algorithms used or algorithms that are not described in the book/handout.
- Justify any major design decisions you made, such as why you chose a particular algorithm or method.
- List any extra work you did on top of basic requirements of which the grader should be aware.

There is also opportunity for up to 10% extra credit by implementing things above the minimum requirements. See section 8 for more details. Particularly impressive projects may be eligible to win a prize.

5 Starter Code

It is recommended that you begin by first reviewing the starter code as provided. Most of the files should be familiar to you by now. The README gives a breakdown of each source file, the most important of which are game/main.cpp, game/flock.cpp/hpp, game/integrator.cpp/hpp.

5.1 What You Need to Implement

game/main.cpp contains partially-filled out functions for initialization, updating, rendering, and event handling. You should modify these to create your
application. We provide a shell of a particle simulation in `game/flock.cpp`, which you may use if you wish, though it is not required.

Feel free to modify any existing code or add new files. You have much more freedom with the behavior this time around. You may also use any code (starter or yours) from any previous labs.

5.1.1 What We Give You

We provide with quite a bit of shell code, if you wish to use it. None of it is required, though we highly suggest your integrator/system interface be similar to the one provided.

- Interfaces for an integrator and system as described in the lecture slides. We provide you with a unimplemented Euler integrator and an example, non-realistic integrator just so you can see how it works. The flock implements the system.
- A shell implementation of the flocking algorithm. None of the forces are implemented, but it provides a base you can start with.
- Event code to move a goal around using the WASD keys.
- An extremely simple (and unacceptable) renderer. You should replace it.

6 Grading: Visual Output and Code Style

Your project will be graded both on the visual output (both screenshots and running the program) and on the code itself. We will read the code.

In this assignment, part of your grade is on the quality of the visuals, in addition to correctness of the math. So make it look nice. Extra credit may be awarded for particularly good-looking projects. See section 8 for more extra credit opportunities.

Part of your grade is dependent on your code style, both how you structure your code and how readable it is. You should think carefully about how to implement the solution in a clean and complete manner. A correct, well-organized, and well-thought-out solution is better than a correct one which is not.

We will be looking for correct and clean usage of the C language, such as making sure memory is freed and many other common pitfalls. These can impact your grade. Additionally, we will comment on your C++-specific usage, though we will generally be more lenient with points. More general style and C-specific style (i.e., rules that apply in both C and C++) will, however, affect your grade.

Since we read the code, please remember that we must be able to understand what your code is doing. So you should write clearly and document well. If the grader cannot tell what you are doing, then it is difficult to provide feedback on your mistakes or assign partial credit. Good documentation is a requirement.
7 Implementation Details

7.1 Particle Systems

7.1.1 Simple Particles

A simple Newtonian particle is an element in space that consists of several parameters in time. For simplicity, we can initially describe a particle using three parameters: position, velocity, and mass. We denote these as \( x, v, \) and \( m \) respectively, where the first two are functions of \( t \), time. Note: for our simple system, we can just consider \( m \) to be constant.

Recall also, Newton’s 2nd Law of Motion, which states \( f = ma \) or \( a = f/m \). Written as a differential equation where \( a \) is the second derivative of \( x \) with respect to time, we have:

\[
\ddot{x} = \frac{f(x, \dot{x})}{m} \tag{1}
\]

As a result, we can now use the following equations to describe the change in state of a particle given the parameters above.

\[
\dot{x} = v \tag{2}
\]

\[
\dot{v} = \frac{f(x, \dot{x})}{m} \tag{3}
\]

At a given time \( t \), the change in \( x \) with respect to \( t \) is simply \( v \). Likewise, the change in \( v \) is some function \( f \) of \( x \) and \( \dot{x} \) over \( m \).

So what is \( f \)? We know that \( f \) is roughly equivalent to the ‘force’ term in Newton’s 2nd Law of Motion. We say that \( f \) takes parameters \( x \) and \( \dot{x} \) because in general, forces are dependant on what would be an particle’s position and velocity. Therefore, forces are basically functions, which describe the effect of forces on an object (e.g. gravity, spring forces, dampening forces, etc).

An example of a force would be gravity. Gravity is a case where \( x \) and \( \dot{x} \) are not parameters, as it is constant with respect to the mass of the object. Instead, gravity is simply some gravitational constant \( g \) multiplied by the mass \( m \):

\[
f_{\text{gravity}} = mg \tag{4}
\]

On every time step, we compute forces acting on our particles. Each particle keeps track of the forces applied to it, using a ‘force accumulator’, which is basically just a sum of the total forces. The parameter \( f \) will now be used to denote this accumulator, which describes total forces acting on the given particle.
7.1.2 Integrators

Particle simulation falls into a category of problems known as initial value problems. Initial value problems are classified by equations of the form:

\[ \dot{x} = f(x, t) \]  \hspace{1cm} (5)

Here \( f \) is some function of \( x \), some state of our system, and \( t \), time. \( \dot{x} \) is the change in \( x \) with respect to time. Basically, in this system, we are given some initial state \( x_0 = x(t_0) \) and wish to follow \( x \) in time, hence the name ‘initial value problem’.

7.1.3 Euler’s Method

One method of solving initial value problems is to find numerical solutions using differential equation solvers, or integrators. As discussed in lecture, the simplest method for this is Euler’s method, which can be described as follows:

\[ x(t_0 + h) = x_0 + h\dot{x}(t_0) \]  \hspace{1cm} (6)

Here we have some initial state \( x_0 = x(t_0) \). We can estimate \( x \) at some later time, which we denote as \( t_0 + h \). The parameter \( h \) is our time step. Thus, we can evaluate \( x(t_0 + h) \) by starting at our initial value and stepping forward by \( h \) in the direction of the derivative.

For this project, implementing Euler integration is sufficient. However, if you are familiar with differential equations, then you probably know that it suffers from both accuracy and instability issues, making it unsuitable for systems requiring large time steps or very tight constraints.

If you are curious or ambitious to try writing a more practical integrator, we discuss better, ‘higher order’ integrators in the appendix of this document (section 10.1).

7.2 Flocking Algorithms

7.2.1 Basic Boids

The first general flocking algorithm was introduced at SIGGRAPH 1987 by Craig Reynolds. The paper describes a method for simulating flocking behavior based on a set of simple rules, which became known as the ‘Boids’ flocking model. A later paper published at GDC 1999 aggregated these rules and more into a general paper for ‘steering behaviors’. We will be implementing four of the rules from this later paper: the original flocking behaviors (separation, alignment, and cohesion) and one additional behavior (goal-seeking).

The term ‘boid’ is used in the paper to refer to a ‘bird-like’ object that follows the flocking behaviors. We will follow this convention through this writeup. In our modified particle system, we will treat each particle as a boid.

Unlike particles, a boid is slightly more complex. A boid also has an orientation, which we will discuss later. Additionally, a boid also has a notion of a
‘neighborhood’ or ‘field of view’. Generally, this region is represented as some sort of volume that encompasses some area directly in front of or around the boid.

For our purposes, we will ignore using complex shapes for field of view and instead just consider a sphere. While this disregards the fact that a boid’s peripheral vision may only go so far naturally, it is an appropriate simplification. For extra credit, you may wish to try implementing more complex representations of field of view.

So in summary, a boid is simply a particle, with the addition of orientation and a field of view. Additionally, since a boid is actually an object and not just a particle (point in space), we may also wish to keep some sort of collision-checking (bounding) object associated with it. The use of this bounding object will be discussed in section 7.2.3.

We will now discuss how these additions will help us to formulate forces to act on the underlying particle system.

7.2.2 Steering Behaviors

Below, we will discuss each of the four steering behaviors that we are asking you to implement, but we will clarify some things first.

One important note is that the algorithm only takes a boid’s ‘local’ knowledge into account. That is, each boid only has knowledge about the flockmates that are within its neighborhood. This makes sense because a single boid has a limited amount of perception as approximated by its field of view and therefore does not have global knowledge about every other member of its flock (unless the flock is fairly small).

Therefore, when applying the forces (steering behaviors), we only consider and look at the neighbors in the field of view. (In practice, this can occasionally result in undesirable effects if a boid is too far from the rest of the flock.) To determine whether or not another boid is in a boid’s field of view, we can do a simple distance check between their positions and the size of the field of view.

7.2.3 Separation

The separation behavior states that a boid will avoid collision with all nearby flockmates that are within its neighborhood. In an actual flock, this behavior manifests itself as what keeps flockmates from crowding together and crashing into each other.

How do consider this in terms of forces? What we want is a force that pushes boids apart if they are too close. However, a boid will always attempt to maintain its distance from its flockmates prematurely, so instead of waiting until they collide, we want to apply repulsive forces based on the distance of a boid to its flockmates. The closer a flockmate is, the more force we should apply and vice-versa.

However, recall that a boid only has a limited, local knowledge and only knows about flockmates in its neighborhood. Therefore, when applying sepa-
ration (and all of the other forces), we should only consider calculating forces relative to the other flockmates in the neighborhood.

You may find object-object intersection useful for computing whether or not boids are in the same neighborhood. We discuss this in the appendix in section 10.2.

### 7.2.4 Alignment

The alignment behavior states that boids will steer towards the average ‘heading’ of their neighboring flockmates. This rule also goes by the term ‘velocity matching,’ which basically describes how we want to formulate this as a force. In an actual flock, this behavior manifests itself as what keeps boids headed in the same direction.

Therefore, for each boid, we can compute the average velocity of all boids in its neighborhood and apply this as a force on the boid. Let $v_i$ be the ith neighbor in a neighborhood of $n$ boids. Therefore, $v_{avg}$ is simply:

$$v_{avg} = \frac{1}{n} \sum_{i=0}^{n} v_i$$  \hspace{1cm} (7)

However, we don’t necessarily want to apply a force in the direction of the average velocity. What we want is a force that will steer us in this direction. One thing to note is that $v_{avg}$ can be represented as a combination of our current velocity $v$ and some other vector that we will call $v_{steer}$.

$$v_{avg} = v + v_{steer}$$  \hspace{1cm} (8)

Interestingly enough, the force we want is just $v_{steer}$, which we can calculate by simply using $v_{avg} - v$.

### 7.2.5 Cohesion

The cohesion behavior states that a boid will steer towards the average position of its flockmates. As the name suggests, this is the behavior that is responsible for keeping the flock together and manifests itself as what keeps the flockmates together as a mass (yet not too close thanks to separation).

In terms of forces, we want to consider cohesion as a force that moves boids in the direction of a particular position, namely the average position of the flockmates. We can call this position $p_{avg}$.

Consider the vector that points in the direction of $p_{avg}$. Let’s call this $d$, where $d$ is just $p_{avg}$ minus the boid’s position $p$.

Now we can do one of two things. It is sufficient to just apply $d$ as our force and be done with cohesion. That is, we can simply push the boid towards the force. However, an equally good way to do this is to apply a force that steers the boid towards this direction. In the same method we handled alignment, we
want $d$ to be the sum of our current velocity $v$ and some steering vector $v_{\text{steer}}$ such that:

$$v_{\text{steer}} = d - v$$

(9)

Note: In practice, both methods work, however, we recommend the latter of the two, as this is what is used by goal-seeking.

7.2.6 Goal-Seeking

The fourth and final rule behavior will cause our flock to actually move in a direction is goal-seeking (note: Reynolds simply defines this as ‘seek’ in his GDC 1999 paper). Without an initial velocity, a flock would not actually go anywhere and simply fluctuate around its starting position. Therefore, goal-seeking is a means of driving the flock.

The idea here is to set some target or goal $g$ which changes over time. The rule then states that all boids will steer and move in the direction of $g$. This is, incidentally, very similar to cohesion, except that we want a force that steers an object in the direction of a target instead of the average position. In this case, our target is $g$.

Note: Simply applying a force in the direction of the goal is insufficient.

7.3 Forces

7.3.1 Force Weights and Dampening Terms

Computing forces to act correctly is tricky and you will probably need some fine-grained tweaking to get your flock to act correctly and realistically.

While not technically a part of the actual physical equations, you may wish to multiply a weighting term (constant) with each of your forces. This term can act either as a scaling or dampening term depending on how much you want a particular force to influence the overall behavior of the flock. For example, separation and obstacle avoidance will likely need very high weighting terms, since we want forces strong enough to push a boid out of an intersection with another object.

Note: It may be tempting to use high scaling/dampening terms. However, thanks to the inaccuracy and instability associated with Euler integration, higher forces mean that it is more likely for your simulation to explode. Start small with your weighting terms.

This may also prove to be fairly useful for debugging, as you can very easily turn off a force by setting its weighting term to zero. As a note, these terms should definitely not be hard-coded into your forces and should be easily changed.

7.3.2 Force Ordering

In addition to having different weights, forces are also subject to an ordering scheme. That is forces of lower precedence should generally be applied first,
since we wish for higher precedence forces to override them if necessary. A sample weighting scheme is given below:

- Cohesion
- Alignment
- Goal-Seeking
- Separation

Here, separation is given the most weight, since collisions should be avoided at all costs, generally at the expense of staying together or heading in the same direction.

It is sufficient to merely sum all of the forces into your force accumulators, but you may wish to have a cleverer method of summing them up. For example, if a boid is too close to crashing into an obstacle, it may be better to simply drop all forces in the direction of the obstacle and apply a force in the opposite direction.

## 7.4 Orientation

As our system stands now, we have basic flocking behavior integrated into our particle system as forces. However, up to this point, we have been making the assumption that all particles are uniform in shape. This assumption fails when we consider boids, as a boid generally has a non-uniform shape. It was previously mentioned that you needed to handle orientation correctly. Particularly you will handle banking, or the change in a boid’s roll.

First consider the boid’s local space, defined by a set of three axes: $x$, $y$, and $z$. In order to establish a boid’s orientation $q$ at a given point in time, we can find these axes, put them in a rotation matrix, and convert to a quaternion.

The z-axis is already given to us. Assuming the boid points down in the direction of the z-axis, then this is simply its normalized velocity $v$.

Next, we want to solve for the boid’s y-axis, or new up-vector. We can do this by first looking at the ‘down-vector’, which is oriented based on a combination of the force of gravity $f_g$ and the amount of outward force due to turning (centrifugal force) $f_c$. Thus, the down-vector $d$, noted in the figure as the resultant acceleration, is simply the combination:

$$d = f_g + f_c;$$ (10)

$f_g$ can be calculated as described in equation 4. $f_c$ is a little trickier. One thing to note is that $f_c$ is orthogonal to $v$. Therefore, given some acceleration in the direction of $v$, we can say that $f_c$ is the component of the force orthogonal to $v$. $f_c$ is given by equation 11.

$$f_c = -(f_t - \hat{v}(\hat{v} \cdot f_t))$$ (11)

Here, $f_t$ is the total force on the object (what is most likely contained in the force accumulator after a timestep) and $\hat{v}$ is just the normalized $v$. What we
are doing here is first calculating the centripetal force and negating it to get the centrifugal force.

So now we have $d$. We can negate this to get $u$, which is our new up-vector. However, this does not account for abrupt changes in roll; as is, the orientation will result in nasty snapping artifacts. We want to damp out these changes and we can do this by adding in some of the old orientation. If $u'$ is the boid’s old vector, we get that our final up-vector $y$ is simply:

$$y = \alpha \ast u + (1 - \alpha) \ast u'$$

Here $\alpha$ is some weight in the range $[0, 1]$ that describes how much banking we want. A higher value for $\alpha$ results in more banking.

Finally, using cross products, we can get $x$ from $y$ and $z$. As a note, $y$ is not guaranteed to be orthogonal to $z$ using this method. To ensure that we have a set of orthogonal axes, we can recompute an orthogonal $y$ by taking an additional cross product using $x$ and $z$.

### 7.5 Rendering

By now, you should be familiar with OpenGL and how to use it to render models to the screen. All that remains once we have a working particle system is to tie it to actual models and a scene, which requires some OpenGL.

Using our particle simulation, we already have nearly everything we need in order to render an entire flock of objects.

We can easily tie a model to a particle. Given a particle (boid) with position $x$ and orientation (stored as a quaternion) $q$, we have the respective translation
and rotation of the object.

For this assignment, we would like to render your boids as models. It does not need to be high resolution (in fact, you should consider low resolution models for large flocks) but it should be non-uniform in shape. We are also giving you the .OBJ file of the fish used in the sample application, which you can use if you wish.

The rest of the scene can be rendered as you like. One suggestion might be to make a simple scene with some blocks or large objects for obstacles. A sky box or sky dome is a quick way to add an environment.

7.6 An Interactive Application

The last requirement is to make your simulation interactive. How you choose to do this is up to you, but it must involve using some keyboard or mouse input. One simple idea is to change the target of the flock using these means.

We are using SDL to handle windows and events. If you are unfamiliar with SDL, you may wish to take a look at the online documentation or some of the setup code from previous projects in order to get a grasp of how this works.

We have already provided you with some events to move the goal with the WASD keys. We highly suggest that you come up with other methods of input.

7.7 Suggested Sequence

We suggest you implement the assignment in the following order:

1. Have a look at the provided code.
2. Implement a basic particle simulator and integrator. Start with Euler integration. We give you the rudiments of an integrator as part of the starter code.
3. Test your simulation by adding some simple forces. For example, you could add gravity and make sure your particles accelerate downwards. For this, you might just want to render spheres or points.
4. Add forces for each of the four behaviors described in the flocking model.
5. Implement changing orientation based on banking.
6. Make your simulation interactive by adding keyboard or mouse input.
7. Consider expanding your simulation into something interesting.

8 Extra Credit

Any improvements, optimizations, or extra features for the project above the minimum requirements can be cause for extra credit, up to 10%. Particularly impressive projects may be eligible to win a prize. Extra credit is generally awarded for impressive achievements beyond the project requirements, at the discretion of the graders.

Ideas may include but are not limited to:
• Create an interesting environment by adding models and scenery.

• Consider making your flocking behavior more complex by adding more advanced behaviors as forces. Some ideas to think about are obstacle avoidance, pursuit, or fleeing behaviors.

• Turn your simulation into an interesting game.

• Write a different integrator for your program. Runge-Kutta 4 is extremely popular in simulations, but you may also wish to investigate one of the following integration schemes: Verlet, Leapfrog, sympletic, and implicit (in order of increasing difficulty to implement).

• Use your particle simulation as the basis for one or more of the following particle-based phenomena: crowds, hair, cloth, fluids, fire, rigid bodies, etc. Course staff can point you in the direction of some resources if you are interested in trying one of these.

• Render your flock in an interesting way (perhaps throw some shaders on for interesting effects). **Warning:** Copying your shaders over from project 3 is fine, but you will not receive any additional points for shaders that have already received extra credit. As in project 3, this does not mean copying some shader that you find online and fiddling with it a little.

9 Words of Advice

9.1 General Advice

• As always, start early. This lab is not particularly difficult, but there is quite a bit that can be done to make it interesting.

• Be conscious about your code. Performance is a large issue for this project.

• Be careful with memory allocation, as too many or too frequent heap allocations will severely degrade performance.

• Make sure you have a submission directory that you can write to as soon as possible. Notify course staff if this is not the case.

• While C has many pitfalls, C++ introduces even more wonderful ways to shoot yourself in the foot. It is generally wise to stay away from as many features as possible, and make sure you fully understand the features you do use.
10 Appendix

10.1 Higher Order Integrators

Recall that Euler’s method has two issues: accuracy and stability. That is to say, given larger time steps or very stiff constraints, it is very likely that our integrator will produce values that cause our system to blow up. While these problems can be mitigated by using ridiculously small time steps, this is impractical for real-time applications, as small time steps do not allow us to run the simulation at a reasonable speed.

The reason for these issues is because Euler’s method does not approximate ‘higher order error’. If we expand out the formula using the Taylor’s series expansion, we get:

\[ x(t_0 + h) = x_0 + h\dot{x}(t_0) + \frac{h^2}{2!}\ddot{x}(t_0) + \frac{h^3}{3!}\dddot{x}(t_0) + \ldots + \frac{h^n}{n!}\frac{\partial^n x}{\partial x^n} \]  

Note that Euler’s method discards all but the first two terms of the expansion (and basically assumes that the rest of the terms evaluate to zero). We describe these discarded terms as the ‘error term’ and denote it as \( O(h^2) \), that is Euler’s method is accurate up to the the second-order term of the Taylor’s expansion. The term ‘nth-order’ is used to describe the order to which an integration scheme is accurate. Thus, Euler’s method is what we would call a ‘first-order’ integrator.

\[ x(t_0 + h) = x_0 + h\dot{x}(t_0) + O(h^2) \]  

Euler’s method is simple and fast to implement, but suffers from the issues we’ve described above. The solution is to consider higher-order integrators, which give us increasing stability and accuracy.

The Midpoint method is a second-order integration scheme that you may be familiar with from your calculus or differential equations courses. It can be described as follows:

\[ x(t_0 + h) = x_0 + h\dot{x}(t_0) + O(h^2) \]  

In this equation we have replaced \( \dot{x} \) with a function \( f \) that evaluates the time derivative at the given \( x \). The method then does the following. It first computes an Euler step at \( x_0 \) and then evaluates the derivative at the midpoint of the step.

The Midpoint method also goes by the name ‘Runge-Kutta 2’ as it belongs to a family of numerical solvers known as ‘Runge-Kutta’ techniques. Runge-Kutta integration provides a formula for any nth-order numerical solution. For all values of \( n < 5 \), nth-order Runge-Kutta requires \( n \) derivative evaluations. For most applications and practical purposes, classic fourth-order Runge-Kutta (RK4) is generally used.
10.2 Object-Object Intersection

This isn’t required for the project, but it may be extremely useful, especially if you are interested in implementing obstacle avoidance.

There are plenty of elegant and excellent ways of computing object-object intersection in a scene. However, given that our objects are dynamic, this rules out methods such as spatial data structures, which are cumbersome to update. Bounding volumes are a fast and simple method of object-object intersection. For this project, it is sufficient to use very simple bounding volumes for your objects. We recommend using bounding spheres or axis-aligned bounding boxes. Conveniently, bounding spheres line up with our notion of a boid’s neighborhood as a sphere.

The following is an example of how to implement bounding spheres and subsequently, how to derive penalty forces based on the amount of intersection.

10.2.1 Bounding Spheres

You may recall from the raytracer project that ray-sphere intersection was fairly straightforward. Likewise, sphere-sphere intersection is also very simple. As spheres are uniform, orientation is not an issue, so what this boils down to is a distance check.

Given two spheres $S_1$ and $S_2$, with radii $r_1$ and $r_2$ respectively, and the calculated distance between them $d$, we know the following.

- If $r_1 + r_2 < d$, then $S_1$ and $S_2$ do not intersect.
- If $r_1 + r_2 = d$, then $S_1$ and $S_2$ intersect at exactly one point.
- If $r_1 + r_2 > d$, then $S_1$ and $S_2$ intersect at multiple points.

10.2.2 Penalty Forces

Once we have confirmed that two objects intersect, it is necessary to compute the forces that will separate the two objects such that they are no longer colliding.

There are a variety of ways to do this, but one simple method that is sufficient for this assignment is penalty forces. As the name suggests, the idea here is to calculate the amount of intersection between two objects and then apply a force in the opposite direction, to ‘push’ the object out to a state where there is no intersection.

If we are using bounding spheres, then this calculation is fairly straightforward. If $d$ represents the distance between two bounding spheres $S_1$ and $S_2$ with radii $r_1$ and $r_2$, then the amount of overlap between them can be described using:

$$d_{\text{overlap}} = d - (r_1 + r_2)$$

Here $d_{\text{overlap}}$ the magnitude of the overlap. Given this, there are several ways to resolve the intersection. You could create a penalty force with magnitude $d_{\text{overlap}}$ that is applied to either $S_1$ or $S_2$ in order to force one out from the other.
Another alternative would be to distribute the amount of overlap, such that two forces are applied, each with magnitude $d_{\text{overlap}}/2$ for a more even distribution.

Of course, recall that force is a vector and has a direction, so you should remember to calculate a vector (or vectors) that points in the reverse direction of the overlap (and ensure that the magnitude is set correctly).