

**Due date:** 9:00am Thursday April 6, 2023. No lates accepted.

**Objectives:** 3D OpenGL animation programming.

**Hand in:** Electronic submission of all source code, executables, and data (see assignment 1). Include cover sheet in submission files.

**Note:** You may work alone or in groups of 2 for this assignment. If in a group, all members must contribute equally to the assignment, and be able to answer questions about all submitted code.

Do **one** of the questions A or B. You should implement the application incrementally, in the suggested order of steps given. Then you should add several the components from the optional list, in any order you wish. Read the general requirements on page 4 before starting.

**General overview:** Both questions involve particle simulations. The first question uses particles in an environment that has a simplified simulation of physics (gravity, collision). The second question uses particles with a flocking engine, to simulate a swarm of agents. Both questions use a similar animation engine described below for processing the particles and updating them within their environments

A particle list is a list of records. Each record represents one entity (particle, agent) in the system. Every time a new entity is created (key stroke, or perhaps automatically every few frames), a new record is created and inserted onto the list. Similarly, when an entity dies, its record is removed from the list. Each record contains all the information about that particle in the environment, and may include some or all of the following:

Position (px, py, pz): 3D coordinate of the position

Direction (dx, dy, dz): the direction vector that the entity is moving towards

Speed (S): higher numbers = faster speeds; 0 = stationary

Rotation angles and angle increments: used for spinning or orienting particles

Scale factors: used for differently scaled particles

Object shape type: used if different shapes/models are possible

Colour (R, G, B): could be for the whole object, or one RGB value per vertex, or a Material, or...

State: a value indicating the behaviour state (eg. foraging, schooling, exploding...)

Age: used if entities have a finite lifespan

(Other special fields...?)

Next Record Pointer

To update a frame of a real-time animation, the following **animation engine** is used:

1. Frame and depth buffers are cleared.
2. Environment (ground, walls, etc.) is rendered
3. For each object in Particle List:
  - a. Update the particle:
    - compute new position, based on environment rules and particle state
    - compute other state variables for particle (kill it if required)
  - b. Render the newly positioned and rotated particle, if it exists

One pass through the list results in all the particles being updated and rendered. The particle update step is dependent upon the kind of simulation being done. When a new particle is created, it will have some initial position in the 3D world, a direction to move towards, and possibly an orientation (rotation) to reflect the direction it is moving. Its position is updated as follows:

Direction = (updated according to rules of environment)

Rotation = (update based on new direction, or random spinning)

Position = Position + Direction\*Speed

The rules for updating the direction are given in A and B below. Speed is optional, and it can be altered as well, which results in accelerating behaviour.

## A. Particle Fountain.

1. Define a large flat polygon to represent the ground. Its center should be at (0,0,0), and lay on the XZ plane. Also define a fountain in the middle (a solid cube, tetrahedron, or any shape you want).
2. A particle takes the shape of a triangle, tetrahedron, small cube, or other shape. It has a record structure as described on p.1. The idea is that a particle resides in the scene at a particular position, and has a particular direction that it is traveling towards. When a particle is created, its start position is at the top of the fountain, with a direction (vx, vy, vz), where the directions are randomized a bit, but are generally in the upwards (positive Y) direction. The particle direction is updated during every frame as follows:

Direction = Direction + (0, -g, 0) : a particle with gravity pulling it down

This expression updates the velocity by applying a small gravitational effect ("-g") to the velocity's Y direction (up-down), making particles eventually fall downwards (-Y direction). You will have to find a suitable value of g through experimentation.

3. The speed can be taken to be 1 by default; if speed S key is invoked, then each particle is given a random speed between 1 and N. This makes some particles fast, and others slow.
4. A particle will collide with the ground when its next Y position goes negative, and its X and Z are within the extents of the ground polygon. To simulate bouncing after a collision, you can simply flip the sign of the Y component of the velocity vector (make it positive).
5. A particle will die when its Y position becomes less than some predetermined negative value. This happens when the particle bounces outside the area of the ground, and falls down the negative Y abyss.
6. There are 3 firing modes:
  - (a) By default, a constant stream of particles shoot out of the fountain.
  - (b) A manual firing option, so that when F is pressed, particles are shot. When F is released, particles cease. Note that you might need to limit the maximum number of active particles.
  - (c) Single shot mode -- just fire 1 particle at a time, per key press.
7. Using another key, toggle a random spin mode to the particles in motion. This requires angle fields such as current angle (Ax,Ay,Az) and angle increments (dax,day,daz) in the particle record. Make sure you don't get over- or under-flow with the angles, by assuring that angles are always between 0 to 360 degrees.
8. Add a friction mode in which particles lose momentum when they collide with the ground. Reduce the speed value by a set amount (eg. 10%). If a particle becomes stationary, delete it.
9. A reset key should reset the entire simulation to the initial configuration.

Options: Do 5 of the following. Do extras for a bonus marks (max 5 items extra for bonus).

10. Can you add trails to particles, like sparks, smoke, or...? Can alpha-blending give a smoke-like effect?
11. Make particles explode! This might happen after a random time span for the particle. It will involve keeping a state field (frame count?). When it is in an exploding state, you will replace the default particle record with new particle record(s), which will perform an exploding effect when rendered frame by frame. The remnants eventually die.
12. "Spray" mode. This controls the degree of randomness of directions assigned to new particles. It toggles between low spray (default, slightly randomness to initial direction) to high spray (wildly random).
13. Add a mode in which sparks are created from any collision of the particle with anything.
- 14.. Add normals to all visible surfaces. Then define lighting and materials for the scene.
15. Make your particles collide with objects on the ground. It will use a similar idea to ground collision, except that a minimum distance between the particle and object center will determine if a collision happens. (You should use "fast" squared distances). Friction should be applied during collisions.
16. Add textures to the surfaces.
17. Add a mode that lets the viewer's eye be one of the particles.
18. Add a mode that gives particles different colours, sizes, and perhaps other unique behaviours.
19. A square hole sits on the ground. Particles will fall right through it.
20. Let particles collide with one another. Friction should be applied. Can you figure out how to make their collision detection efficient, and behave realistically?
21. Add sound FX.
23. Add your own idea!

**B. Flocking agents:** This question simulates Artificial Life agents swarming in a virtual environment.

1. Create a polygonal model for a simple world. It should be a volume with rectangular sides. It should be centered on the screen, with (0, 0, 0) in the center. A large polygon represents the ground, and lines should define the world edges (nothing should leave the arena!) Feel free to embellish the world with objects.

2. Create a simple agent model to denote a particle. You can make the agent abstract, and represent it with a polygonal model. It should be oriented in the direction that it is moving (see example program "orient.c" at end of assignment). If you like, you can create a theme for your flocking simulation: birds, insects, fish, UFOs... Feel free to be creative in your modeling, with appropriate models for the agents, environment, etc.

3. Each agent has a data structure that records all the relevant information about itself (see general discussion). One special agent is designated the leader. The leader is initialized with a random Position and Destination, and possibly Speed. When it reaches it (i.e. gets very close to it), it finds a new random destination on the ground (X-Z plane). The destination is always within the world extents.

4. The other agents have 2 basic states (user-selectable). The default state is exploration, in which they move independently, in the same manner as the leader. The other mode is swarming, where each agent takes the leader's current position as being the destination. Their direction is determined by the leader's current position:

$$\text{Direction} = (\text{Leader Position} - \text{Agent Position}) \quad (\textit{normalized})$$

5. There are some "flocking rules" that are used when agents are in swarming mode:

a) *Destination rule:* Each agent's destination is the leader's current location.

b) *Breathing-space rule:* An agent never likes to be closer than a minimal distance D1 from its nearest neighbour. When the closest agent is within D1, it will try to move away from it (eg. change the destination to be in a direction opposite from the nearest entity).

c) *Socialability rule:* An agent never likes to be more than a distance D2 from the nearest entity.

When the nearest entity is more than D2, a agent will want to move towards it. Note that  $D1 < D2$  always.

d) *Prime Directive:* An agent can never leave the arena's world extents.

Note that rules b and c take precedence overrule a; but when b and c are satisfied, then rule a is used.

Rule d is always used.

6. Speed is a constant by default, but can be toggled to a random value with key S.

7. The user should be able to generate multiple agents. Every time a key is pressed, a new agent appears. There is an "agent list" with all the records. To do the simulation, each entity record is updated as above, and then drawn. Agents should die too (toggle a max life span?).

8. Don't let agents collide with each other. If the next position of a agent takes it too close to another, have it stop (don't update its position or velocity). Alternately, you may want to move it in the opposite direction as the current velocity briefly, to avoid a collision. Another option is to have it move backwards, but with some random perturbation. This makes the agents reverse and sway away.

Options: Do 5 of the following. Do extras for a bonus marks (max 5 items extra for bonus).

10. Make your agent change colour when it is in different states: too close to an agent, too far away, following, swarming mode, flocking mode, etc. This is a great debugging tool.

11. Add a mode that adds smooth turning to the movement. Instead of agents instantly turning towards new destinations, each will gradually turn towards a new destination, resulting in a more realistic, smoother movement. It is accomplished with:

$$\text{Direction} = \text{Direction} + A * (\text{Destination} - \text{Position})$$

where A is a small fraction. The new direction should be normalized. Thus when a agent reaches its destination, and a new destination is determined, it will gradually turn towards this new destination. Be aware that sometimes agents may circle around a destination without reaching it (they're in orbit around it!). Can you figure out a solution for that?

12. Add normals to all visible surfaces. Then define lighting and materials for the scene.

13. Add some additional social behavior rules. For example, mating or predator/prey behaviour might be interesting.

14. Add textures to the surfaces.

15. Add a mode that lets the viewer's eye be one of the agents, giving an agent's perspective of life in the swarm.
16. Replace the constant Speed in 6 with a variable Speed (new field in the entity record). It will go between 0 and some maximum value, with some constant increment. When an entity reaches its destination, its speed will be reduced.
17. When toggled, a "trail mode" shows the previous locations (paths) of agents in the form of dots.
18. When an agent dies, it explodes.
19. Add sound FX.
20. Add your own idea!

**General requirements:** Both A and B should incorporate the following:

- (i) Draw the model such that it is centered on the screen. For camera rotations below to work, the ground polygon should be centered about the origin.
- (ii) Use "gluLookAt" to move your eye towards and above the positive z-axis so that you have a good view of the object. Use "glPerspective" to incorporate perspective and scale. Experiment with the parameters of these commands to find a good image (interactive access to the parameters is very beneficial!).
- (iii) Add the option of plotting the objects (particles) as a single vertex (GL\_POINTS), wireframe (GL\_LINES) or solid surface (GL\_POLYGON). An appropriate command key or menu should select amongst these modes.
- (iv) Toggle between flat shaded and Gouraud shaded solid polygons (another key). You will only see this effect if your vertices have different colours, or lighting is used.
- (v) Use OpenGL's backface culling to remove back-facing polygons. Note that, depending on the parameter settings, if you draw polygons in the wrong direction, you will get holes in your surfaces! Have a key toggle backface culling on or off.
- (vi) Let the user interactively spin the scene about the y-axis, and animate this spinning. The left mouse button should increase rotation in one direction, while the right button increases rotation in the other direction. When the user types x, y, or z, then the mouse buttons will control rotation around that axis. Include a reset (R key) that stops all rotation, and resets the orientation to its original position. (see rotate2.c below). Also, you might wish to use the mouse to rotate your view as well.
- (vii) Use double buffering for all animation.
- (viii) Whenever possible, make all features and options in your program user-selectable. You should consider using GLUT menus. Command keys are fine as well.
- (ix) Print out all the user command keys options on the text window.
- (x) Normals should not be shared amongst vertices (relevant for lighting).

### **Example programs**

3D rotation control with lighting and normals (for a cube):

[www.cosc.brocku.ca/Offerings/3P98/course/OpenGL/3P98Examples/OpenGLExamples/rotate\\_light.c](http://www.cosc.brocku.ca/Offerings/3P98/course/OpenGL/3P98Examples/OpenGLExamples/rotate_light.c)

Setting up orientation of swarming objects:

[www.cosc.brocku.ca/Offerings/3P98/course/OpenGL/3P98Examples/OpenGLExamples/orient.c](http://www.cosc.brocku.ca/Offerings/3P98/course/OpenGL/3P98Examples/OpenGLExamples/orient.c)

GLUT menus:

[http://www.cosc.brocku.ca/Offerings/3P98/course/OpenGL/misc/glut\\_column1.pdf](http://www.cosc.brocku.ca/Offerings/3P98/course/OpenGL/misc/glut_column1.pdf)

Running examples of flocking and particles: (see [3P98 gallery page](#))

Audio FX? Students have recommended libsndfile...

<http://www.mega-nerd.com/libsndfile/>