CST8234 – C Programming
Assignment: Particle System

Objectives

Design, code and test a simple implementation of linked lists
Use multiple files in a program
Use of a Makefile

Submission

To BB in a zip file with the following specifications:

```
LastName_LastName_02_CST8234.zip
```

When untaring the file, it should create the following directory structure:

```
Ayala_02_CST8234
+
+--------> Ayala_particles.c
+--------> Ayala_particles.h
+--------> Ayala_particle.c
+--------> Ayala_particle.h
+--------> Ayala_ANY_OTHER_FILE.c
+--------> Ayala_ANY_OTHER_FILE.h
+--------> Ayala_particles.txt
```

`Ayala_particles.txt`

A small document explaining the type of particle system you created. You need to talk about your particle attributions, creation of particles and methods used to transform you particles.

You should include a paragraph(s) with timing result for your particle system. You are to time you algorithms with different number of particles (for example, start with 500, go to 2000, and then to 10000) and report the time that it takes to create and animate your system.

Assignment Specification

A particle system is a computer graphic technique use to simulate certain fuzzy objects, such as fire, clouds, rain, snow, explosion, etc.

A particle system is composed of one or many entities, the particles. To animate them you need a set of rules to obtain the effect you want.

You are to implement a particle system, using a single linked list. A particle could be defined as:

```
struct particle {
   Color4              color;
   Point3D             pos;
   Vector3D            dir;
   Vector3D            spd;
   int                 lifespan;
   int                 size;
   struct particle*  next;
};
```

The `color, Color4,` is defined as a RGBA values, Red, Green, Blue and Alpha, number from 0.0 – 1.0, that represents the intensity of the color. For example, to represent a very intense red, (1.0, 0.0, 0.0, 1.0). The alpha value represents how opaque the color is (transparency), where 1.0 means not opaque at all, and values closest to 0.0, means the object is very transparent.

```
struct color {
   float r;
   float g;
   float b;
   float a;
};
typedef struct color Color4;
```

The `pos, Point3D,` is the position of the particle in your 3D coordinate system. The `dir` and `spd`, represent the direction of the particle and the speed of the particle respective. If you want your particle to move upwards, it should have a direction (0.0, 1.0, 0.0) – The vector up in the Y-coordinate. Now, if you want your particle to move a speed of 5 positions / frame, then the speed should be (5.0, 5.0, 5.0). To calculate the next position of your particle, you need to multiply the speed by the direction.

```
struct vector
{
   float x;
   float y;
   float z;
};
typedef struct vector Point3D;
typedef struct vector Vector3D;
```

Basic Linked List Implementation

You are to write the following functions:

```
/*********************************************************************
/* FUNCTION :  particle_init
/* PURPOSE  :  initialize the properties of a single particle
/* INPUT    :  pointer to the particle structure to be initialized
/* OUTPUT   :  returns -1 on error, 0 on success
/* NOTES    :
 *********************************************************************/
int particle_init(struct particle* p);
```

```
/*********************************************************************
/* FUNCTION :  particle_add
/* PURPOSE  :  add a particle to the dynamic particle linked list
/* INPUT    :  struct particle *head. Head of the particle list
/* OUTPUT   :  returns -1 on error, 0 on success
/* NOTES    :  Calls particle_init()
 ********************************************************************/
 int particle_add( struct particle **head);
```

```
/*********************************************************************
/* FUNCTION :  particle_remove
/* PURPOSE :   remove a specific particle from the dynamic
               particle linked list
/* INPUT    :  pointer to the particle to remove
/* OUTPUT   :  returns -1 on error, 0 on success
/* NOTES    :  Particle can be situated in any place in the list.
/*             Usually deleted because the lifespan ran out
 ********************************************************************/
int particle_remove(struct particle* p);
```

```
/*********************************************************************
/* FUNCTION :  particle_destroy
/* PURPOSE  :  free memory used by the dynamic particle linked list
/* INPUT    :  struct particle **head. Head of the particle list
/* OUTPUT   :  returns -1 on error,
               the number of particles destroyed on success
/* NOTES    :  removes all particles from the list
               Calls particle_remove()
 ********************************************************************/
int particle_destroy( struct particle **head );
```

```
/********************************************************************
/* FUNCTION :  particle_update
/* PURPOSE  :  update the particles properties to be rendered
               in the next frame
/* INPUT    :  struct particle **head. Head of the particle list
/* OUTPUT   :  returns -1 on error, 0 on success
/* NOTES    :  Creativity and more creativity here for you !!!
 ******************************************************************* /
int particle_update(struct particle **head );
```

Transforming the particle
Some of the particles' attributes are very simple to transform, for example: to update the life_span, you could just decrement the value by a preset value:

```
      p->life_span -= DELTA_LIFE_SPAN;
```

other attributes are more complex to transform. For example, to move a particle you need to (depends on the type of particle system you want to create)

- movement because own forces

```
  particle->pos += particle->pos + (particle->speed * particle->direction )
```

- movement due gravity – acceleration towards down.

```
  Particle->posi += particle->pos + particle->( 31 feet / sec ) * ( 0, -1,
  0 );
```

- movement due collision with objects: Reduce the speed and bounce in a different direction – notice that this depends on what type of phenomenon you are trying to simulate.