Universidad Don Bosco
Escuela de Computación
Trabajo de Graduación

MANEJADOR DE SISTEMAS DE PARTÍCULAS

Presentado Por: Juan Carlos Cruz Dada CD-980202

Asesor: Ing. Oscar Durán Vizcarra

Tutor: Lic. Mauricio Coto

Evaluadores: Ing. Enrique Aguilar Acosta
Ing. Juan Carlos Castro
Lic. Santiago Abarca

Soyapango, 9 de Septiembre de 2003

INDICE

TEMA

Manejador de Sistemas de Partículas

JUSTIFICACION

En gráficas generadas por computadora, existen objetos difíciles de representar como una colección de primitivas de superficies, aún con las ventajas de los mapas de textura y transparencias. Estos objetos incluyen fenómenos naturales como humo, nubes, fuego y agua, entre otros. Es posible representar dichos objetos a través de un sistema de partículas.

Un sistema de partículas está definido como una colección de entidades, relacionadas entre sí o no, que se comportan de acuerdo a un conjunto de reglas lógicas preestablecidas.

Un manejador de sistemas de partículas se encarga de optimizar el uso de estos sistemas, para minimizar la cantidad de memoria utilizada y poder facilitar el control de los sistemas.

Internacionalmente se pueden encontrar tutoriales sobre los sistemas de partículas, pero la elaboración de éstos no está orientada a objetos ni facilitan la optimización de los mismos para animaciones en tiempo real.

En El Salvador no hay fuentes conocidas para la búsqueda de información acerca del desarrollo de sistemas de partículas, ni de los manejadores de sistemas de partículas, por lo que el siguiente trabajo generará una base de investigación y desarrollo en el área de simulación gráfica, y proporcionará ejemplos para el desarrollo de aplicaciones de este género, utilizando la programación orientada a objetos.

Descripción

En este trabajo se presentarán las partes de los sistemas de partículas (entidades y reglas lógicas), sus atributos y las formas de ser representados utilizando conceptos de programación orientada a objetos, todo ello encaminado a la elaboración de un manejador de sistemas de partículas estable y fácil de adaptar a una máquina de modelado en tiempo real (*real time rendering machine*).

Objetivos

Objetivo General

- Crear un manejador de sistemas de partículas que pueda controlar múltiples sistemas y pueda ser reutilizado en el desarrollo de aplicaciones de simulación en 3D (Tres Dimensiones).

Objetivos Específicos

1. Demostrar las diferentes aplicaciones de los sistemas de partículas.

2. Crear una base (FrameWork) que pueda ser implementada en múltiples plataformas, con cambios mínimos en su código fuente.

3. Implementar la animación de algunos sistemas de partículas en tiempo real de manera realista.

4. Desarrollar el sistema utilizando programación orientada a objetos, para facilitar su reutilización y expansión.

5. Usar eficientemente la memoria, mediante el manejador de sistemas de partículas.

Alcances

- ***Portabilidad:*** El sistema podrá ser utilizado en múltiples plataformas, Linux y Windows, con el cambio mínimo de código.

- ***Uso Eficiente de Memoria:*** Al poder reutilizar los sistemas de partículas, se ahorrará memoria.

- ***Reutilización:*** El sistema podrá ser adaptado fácilmente a cualquier otro proyecto en desarrollo para utilizar los sistemas de partículas.

- ***Expansión:*** Al ser desarrollado utilizando la programación orientada a objetos, permite su expansión y modificación según el usuario lo necesite.

Limitaciones

- El número máximo de sistemas de partículas que podrán coexistir en un mundo virtual está determinado en gran medida al hardware que posea el equipo en que se realice la simulación.

- La aplicación no tendrá detección de colisiones.

- Se desarrollará el software de ejemplo en base a las capacidades del sistema de desarrollo.

INTRODUCCION

Los sistemas de partículas son una colección finita de elementos independientes, regidos por ciertas leyes que aplicadas en conjunto permiten generar fenómenos naturales complejos y capturan la vista de quien los percibe, yendo desde efectos tipo caricatura hasta efectos de gran realismo.

En las siguientes páginas, el lector encontrará la teoría básica necesaria para la elaboración de un sistema de partículas, y un framework para el desarrollo de un manejador.

La investigación se ha realizado con el fin de proporcionar un elemento de consulta y ayuda para la creación de sistemas de partículas, ya que en el medio nacional hay muy poca información al respecto.

El documento se divide en tres partes: marco teórico, en el cual se describen los sistemas de partículas, sus métodos y elementos principales; **diseño e implementación de sistemas de partículas**, donde se presenta la implementación de algunos de los sistemas más comunes, como la lluvia, el humo, la nieve, las explosiones y los fuegos artificiales. Por último se encuentra el manejador de sistemas de partículas, el cual presenta ciertas modificaciones a los sistemas para poder ser utilizados minimizando el código y optimizando su utilización.

1. MARCO TEORICO

1.1 Sistemas de partículas

Un sistema de partículas es una colección finita de elementos individuales o partículas, que juntas representan un fenómeno natural, por ejemplo nieve, lluvia o humo. Cada partícula posee atributos individuales como color, energía, posición, etc. y actúa de manera autónoma, pero relacionada con las demás mediante las reglas de modelado y actualización del sistema, por medio de las cuales representan el fenómeno deseado. A través del tiempo, el sistema genera nuevas partículas, las cuales se mueven y cambian dentro del mismo, hasta morir.

Para calcular cada *frame*[1] en una secuencia, tradicionalmente se siguen los siguientes pasos:

a) Generación de partículas: nuevas partículas son generadas en el sistema.

b) Inicialización de partículas: cada partícula obtiene sus atributos iniciales.

c) Eliminación: toda partícula que ha agotado su energía (ha pasado el periodo de vida asignado por el sistema) es removida del mismo.

d) Actualización: se aplican las reglas de movimiento y transformación a las partículas restantes.

e) *Render*[2]: las partículas son dibujadas en pantalla.

1.1.1 Generación de partículas

[1] Cuadro completamente dibujado y listo para ser mostrado en pantalla.
[2] Esta palabra no posee una traducción específica al español, y podría conceptualizarse como "presentación en pantalla". Para facilitar el desarrollo del documento, y evitar significados erróneos, se utilizara el concepto en inglés en el documento.

Las partículas son generadas mediante procesos estocásticos controlados. Uno de éstos debe controlar el número de partículas que deben ingresar al sistema en cada intervalo de tiempo. El número de partículas es importante, porque de él depende la densidad del sistema, y la memoria consumida por el mismo.

1.1.2 Inicialización de partículas

El diseñador elige el proceso que el sistema utilizará para determinar los valores de los atributos para cada nueva partícula generada. En este caso, se ha utilizado una generación aleatoria que oscila entre rangos definidos mediante prueba y error, seleccionando los que proporcionen al sistema el mayor realismo en la animación. Algunos de los atributos asignados son:

a) Posición. Es necesario saber la ubicación de la partícula con el objetivo de poder presentarla en pantalla correctamente. Para algunos efectos, también se utiliza la posición anterior de la partícula.

b) Velocidad. Almacena el cambio de posición respecto al tiempo de la partícula. Se almacena en un vector para facilitar la velocidad y la dirección de dicha partícula.

c) Tamaño. Normalmente se maneja un tamaño estándar para las partículas, pero hay ciertos sistemas que utilizan tamaños independientes para cada una. Por esta razón, también se suele almacenar el cambio de tamaño de la partícula en función del tiempo.

d) Color. Almacena el color de cada partícula. Si el sistema cambia el color de las partículas respecto al tiempo, se almacena dicho valor de cambio.

e) Peso. Determina la forma en que afectarán las fuerzas externas a la partícula.

f) Energía. Sirve para mantener activa una partícula por un periodo de tiempo determinado.

1.1.3 Eliminación

Cada vez que una partícula es creada, se le asigna energía, la cual irá disminuyendo a medida que pase el tiempo. Cuando su energía se agota, esa partícula muere y es eliminada del sistema.

Hay otras formas de eliminar una partícula; esto varia de sistema a sistema, por ejemplo, si una partícula contiene una transparencia total, de nada sirve actualizarla o mantenerla viva, pues ya no ayuda en nada a la imagen que se está generando; así también, si la partícula sale de ciertos rangos preestablecidos, conocidos como umbral, ésta puede ser eliminada antes que se agote su energía.

1.1.4 Actualización

Cada partícula cambia sus atributos a medida que transcurre el tiempo. Por ello, el sistema de partículas posee atributos que ayudan a esta modificación para llegar a la representación del fenómeno deseado.

Según sea el sistema, éste puede contener una gran gama de ecuaciones para calcular movimiento, o sencillamente utilizar una variable de aceleración con la cual se podrá modificar la velocidad de una partícula, y así dar el efecto de existencia de gravedad.

1.1.5 Render

Una vez se ha actualizado el estado de las partículas, el siguiente paso es presentarlas en pantalla. Para facilitar el algoritmo de rendering, se han asumido ciertos criterios:

a) Las partículas no colisionan con otros objetos en la escena.

b) Las fuerzas que actúan sobre las partículas serán únicamente las fuerzas definidas por el sistema.

c) El suelo es plano, paralelo al eje Y, ubicado en Y = 0.

1.1.6 Atributos

Las propiedades mínimas que debe tener un sistema de partículas son:

a) Origen: punto de ubicación del sistema.

b) Número máximo de partículas: contendrá el número de partículas que puede tener activas el sistema.

c) Número de partículas existentes: contendrá la cantidad de partículas activas del sistema.

d) Tiempo acumulado: tiempo que el sistema tiene en existencia.

e) Fuerzas: Sumatoria de fuerzas que afectan al emisor de partículas del sistema, las cuales podrían afectar también a las partículas del sistema.

f) Lista de partículas: Lista que contendrá las partículas pertenecientes al sistema.

2. DISEÑO E IMPLEMENTACION DE LOS SISTEMAS

2.1 Partículas

Como se ha visto, cada partícula es visualizada como un punto en el espacio, el cual indicará su posición junto con otros atributos adicionales. A continuación presentamos la *estructura de la partícula*[3]:

```
struct particle_t
{
  CVector  m_pos;
  CVector  m_prevPos;
  CVector  m_velocity;
  CVector  m_acceleration;
  float   m_energy;
  float   m_size;
  float   m_sizeDelta;
  float   m_weight;
  float   m_weightDelta;
  float   m_color[4];
  float   m_colorDelta[4];
};
```

Código 1: estructura que representa una partícula.

Esta estructura permitirá almacenar las variables mínimas necesarias para una partícula, según lo descrito en el marco teórico.

2.2 Sistemas de Partículas

Los sistemas de partículas serán representados por una clase genérica, la cual contendrá los atributos y métodos que un sistema de partículas necesita. Estos métodos tendrán las inicializaciones básicas para los sistemas de partículas genéricos, los cuales podrán ser modificados por las diferentes implementaciones del mismo.

[3] Debido a que las funciones de OpenGL y GLUT están en inglés, el código se desarrollará en inglés. Para facilitar su lectura, todo el código fuente será presentado en letra `COURRIER`

```
class CParticleSystem
{
public:
  CParticleSystem(int maxParticles, CVector origin);
  virtual void  Update(float elapsedTime)     = 0;
  virtual void  Render()                      = 0;
  virtual int   Emit(int numParticles);
  virtual void  InitializeSystem();
  virtual void  KillSystem();
protected:
  virtual void InitializeParticle(int index) = 0;
  particle_t *m_particleList;
  int         m_maxParticles;
  int         m_numParticles;
  CVector     m_origin;
  float       m_accumulatedTime;
  CVector     m_force;
};
```

Código 2: clase que representa un sistema de partículas genérico, con sus atributos y métodos (funciones) básicos.

2.2.1 Métodos

Los métodos del sistema de partículas contendrán los pasos necesarios para la animación de las partículas, Algunos de los métodos abarcan más de un paso, y algunos pasos han sido divididos en varios métodos.

2.2.1.1 CParticleSystem

Este método se encarga de inicializar los atributos del sistema de partículas, y es llamado automáticamente al crear una clase. Recibe como parámetros el número máximo de partículas que poseerá el sistema y su posición de origen.

```
CParticleSystem::CParticleSystem(int maxParticles, CVector origin)
{
  m_maxParticles = maxParticles;
  m_origin = origin;
  m_particleList = NULL;
}
```

Código 3: constructor de la clase del Sistema de Partículas, encargado de inicializar los atributos el sistema.

2.2.1.2 InitializeSystem

Se encarga de completar la inicialización del sistema, creando la lista de partículas del sistema con el número máximo de partículas que el sistema podrá utilizar, y llevando los atributos de partículas utilizadas y tiempo acumulado a cero.

```
void CParticleSystem::InitializeSystem()
{
  if (m_particleList)
  {
    delete[] m_particleList;
    m_particleList = NULL;
  }
  m_particleList = new particle_t[m_maxParticles];
  m_numParticles = 0;
  m_accumulatedTime = 0.0f;
}
```

Código 4: termina la inicialización del sistema de partículas.

La inicialización del sistema se ha separado en dos métodos para facilitar la reinicialización de un sistema con los mismos atributos iniciales.

2.2.1.3 Emit

Este método intenta generar el número de partículas deseado, siempre y cuando no sobrepase el número máximo de partículas que el sistema puede utilizar.

```
int CParticleSystem::Emit(int numParticles)
{
  while (numParticles && (m_numParticles < m_maxParticles))
  {
    InitializeParticle(m_numParticles++);
    --numParticles;
  }
  return numParticles;
}
```

Código 5: genera el número solicitado de partículas, siempre que no exceda el número máximo.

Emit regresa el número de partículas que no pudo crear, esto para facilitar el manejo de errores en caso sea necesario.

2.2.1.4 KillSystem

Al ser borrado el sistema, es necesario liberar recursos; este método se encarga de liberar los recursos utilizados.

```
void CParticleSystem::KillSystem()
{
  if (m_particleList)
  {
    delete[] m_particleList;
    m_particleList = NULL;
  }
  m_numParticles = 0;
}
```

Código 6: Devuelve los recursos utilizados al sistema.

2.2.1.5 Render, Update, InitializeParticle

Estos métodos contienen las reglas propias de cada sistema y no tienen implementación genérica.

2.2.2 Implementación de diversos sistemas de partículas

A Continuación se presentan generalidades para la implementación de sistemas de partículas:

a) Los métodos InitializeSystem y KillSystem han sido modificados para aceptar ciertos parámetros adicionales y cargar la textura que el sistema proporcionará a sus partículas.

```
glGenTextures(1, &m_texture);
glBindTexture(GL_TEXTURE_2D, m_texture);
BITMAPINFOHEADER bitmapInfoHeader;
unsigned char *buffer = LoadBitmapFileWithAlpha("textura.bmp",
  &bitmapInfoHeader);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
  GL_LINEAR_MIPMAP_NEAREST);
glTexImage2D(GL_TEXTURE_2D, 0, 4, bitmapInfoHeader.biWidth,
  bitmapInfoHeader.biHeight, 0, GL_RGBA, GL_UNSIGNED_BYTE,
  buffer);
gluBuild2DMipmaps(GL_TEXTURE_2D, 4, bitmapInfoHeader.biWidth,
  bitmapInfoHeader.biHeight, GL_RGBA, GL_UNSIGNED_BYTE, buffer);
free(buffer);
CParticleSystem::InitializeSystem();
```

Código 7: Código de Inicialización del sistema cargando una textura.

```
  if (glIsTexture(m_texture))
  {
    glDeleteTextures(1, &m_texture);
  }
  CParticleSystem::KillSystem();
```

Código 8: Código de eliminación de sistema, borrando la textura.

b) La fórmula a utilizar en los fenómenos que requieran emisión continua de partículas será

Ecuación 1	**NP = PPS * Δt** donde, NP: nuevas partículas PPS: partículas por segundo Δt: tiempo transcurrido

la cual cálcula el número de partículas que el sistema deberá generar según el diferencial de tiempo transcurrido.

c) Las velocidades iniciales de las partículas serán calculadas en base a

Ecuación 2	**V_0 = Vpm + rand() * ΔV** donde, V_0: velocidad inicial Vpm: velocidad promedio rand(): función que devuelve valores aleatorios entre -1 y 1. ΔV: variación de la velocidad promedio.

2.2.2.1 Nieve

2.2.2.1.1 InitializeParticle

Inicializa los copos de nieve ubicándolos dentro del área del emisor que es un cuadrado. Todos los copos de nieve tienen un tamaño constante.

```
void CSnowstorm::InitializeParticle(int index)
{
  m_particleList[index].m_pos.y = m_height;
  m_particleList[index].m_pos.x = m_origin.x + FRAND * m_width;
  m_particleList[index].m_pos.z = m_origin.z + FRAND * m_depth;
  m_particleList[index].m_size = SNOWFLAKE_SIZE;
  m_particleList[index].m_velocity.x = SNOWFLAKE_VELOCITY.x + FRAND
    * VELOCITY_VARIATION.x;
  m_particleList[index].m_velocity.y = SNOWFLAKE_VELOCITY.y + FRAND
    * VELOCITY_VARIATION.y;
  m_particleList[index].m_velocity.z = SNOWFLAKE_VELOCITY.z + FRAND
    * VELOCITY_VARIATION.z;
}
```

Código 9: Inicializa los copos de nieve.

2.2.2.1.2 Update

En este método se aplican las reglas que rigen la animación de la nieve:

a) Primero, se actualiza la posición de los copos de nieve en base a la velocidad y al tiempo transcurrido desde la última actualización.

b) Se verifica el estado del copo de nieve. Si ya ha caído al suelo, la partícula muere.

c) Se calculan cuantas partículas deberán emitirse.

d) Se actualiza el tiempo acumulado en base al número de emitidas.

e) Se emiten las nuevas partículas.

```
void CSnowstorm::Update(float elapsedTime)
{
  for (int i = 0; i < m_numParticles; )
  {
    m_particleList[i].m_pos = m_particleList[i].m_pos +
      m_particleList[i].m_velocity * (elapsedTime –
      m_particleList[i].m_startTime);
    if (m_particleList[i].m_pos.y <= m_origin.y)
    {
      m_particleList[i] = m_particleList[--m_numParticles];
    }
    else
    {
      ++i;
    }
  }
  m_accumulatedTime += elapsedTime;
  int newParticles = SNOWFLAKES_PER_SEC * m_accumulatedTime;
  m_accumulatedTime -= 1.0f/(float)SNOWFLAKES_PER_SEC * newParticles;
  Emit(newParticles);
}
```

Código 10: Actualiza los copos de nieve, aplicando las reglas del sistema.

2.2.2.1.3 Render

```
void CSnowstorm::Render()
{
  glEnable(GL_BLEND);
  glEnable(GL_TEXTURE_2D);
  glBlendFunc(GL_SRC_ALPHA, GL_ONE);
  glBindTexture(GL_TEXTURE_2D, m_texture);
  glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
  CVector partPos;
  float size;

  glBegin(GL_QUADS);
  for (int i = 0; i < m_numParticles; ++i)
  {
       glColor3f(1.0,1.0,1.0);
    partPos = m_particleList[i].m_pos;
    size = m_particleList[i].m_size;
    glTexCoord2f(0.0, 1.0);
    glVertex3f(partPos.x, partPos.y, partPos.z);
    glTexCoord2f(1.0, 1.0);
    glVertex3f(partPos.x + size, partPos.y, partPos.z);
    glTexCoord2f(1.0, 0.0);
    glVertex3f(partPos.x + size, partPos.y - size, partPos.z);
    glTexCoord2f(0.0, 0.0);
    glVertex3f(partPos.x, partPos.y - size, partPos.z);
  }
  glEnd();
  glDisable(GL_BLEND);
  glDisable(GL_TEXTURE_2D);
}
```

Código 11: Presenta los copos de nieve en pantalla.

2.2.2.2 Lluvia

2.2.2.2.1 InitializeParticle

Inicializa las gotas de lluvia ubicándolas dentro del área del emisor que es un cuadrado. Todas las gotas de lluvia tienen un tamaño constante, y su velocidad varia únicamente con respecto al eje Y.

```
void CRain::InitializeParticle(int index)
{
  m_particleList[index].m_pos.y = m_height;
  m_particleList[index].m_pos.x = m_origin.x + FRAND * m_width;
  m_particleList[index].m_pos.z = m_origin.z + FRAND * m_depth;
  m_particleList[index].m_size = DROP_SIZE;
  m_particleList[index].m_velocity.x = DROP_VELOCITY.x ;
  m_particleList[index].m_velocity.y = DROP_VELOCITY.y +
    FRAND * 0.5;
  m_particleList[index].m_velocity.z = DROP_VELOCITY.z ;
  m_particleList[index].m_startTime = elapsedTime;
}
```

Código 12: Inicializa las gotas de lluvia.

2.2.2.2.2 Update

En este método se aplican las reglas que rigen la animación de la lluvia:

a) Primero, se actualiza la posición de las gotas de lluvia en base a la velocidad y al tiempo transcurrido desde la última actualización.

b) Se verifica el estado de la gota de lluvia. Si ya ha caído al suelo, la partícula muere.

c) Se calculan cuantas partículas deberán emitirse.

d) Se actualiza el tiempo acumulado en base al número de partículas emitidas.

e) Se emiten las nuevas partículas.

```
void CRain::Update(float elapsedTime)
{
  for (int i = 0; i < m_numParticles; )
  {
    m_particleList[i].m_pos = m_particleList[i].m_pos +
      m_particleList[i].m_velocity * elapsedTime;
      m_particleList[i].m_startTime = elapsedTime;
    if (m_particleList[i].m_pos.y <= 0)
    {
      m_particleList[i] = m_particleList[--m_numParticles];
    }
    else
    {
      ++i;
    }
  }
  m_accumulatedTime += elapsedTime;
  int newParticles = DROPS_PER_SEC * m_accumulatedTime;
  m_accumulatedTime -= 1.0f/(float)DROPS_PER_SEC * newParticles;
  Emit(newParticles);
}
```

Código 13: Actualiza las gotas de lluvia, aplicando las reglas del sistema.

2.2.2.2.3 Render

```
void CRain::Render()
{
  glEnable(GL_BLEND);
  glEnable(GL_TEXTURE_2D);
  glBlendFunc(GL_SRC_ALPHA, GL_ONE);
  glBindTexture(GL_TEXTURE_2D, m_texture);
  glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
  CVector partPos;
  float size;
  glBegin(GL_QUADS);
  for (int i = 0; i < m_numParticles; ++i)
  {
      glColor4f(0.0,0.0,1.0,0.6);
    partPos = m_particleList[i].m_pos;
    size = m_particleList[i].m_size;
    glTexCoord2f(0.0, 1.0);
    glVertex3f(partPos.x, partPos.y, partPos.z);
    glTexCoord2f(1.0, 1.0);
    glVertex3f(partPos.x + size, partPos.y, partPos.z);
    glTexCoord2f(1.0, 0.0);
    glVertex3f(partPos.x + size, partPos.y - size, partPos.z);
    glTexCoord2f(0.0, 0.0);
    glVertex3f(partPos.x, partPos.y - size, partPos.z);
  }
  glEnd();
  glDisable(GL_BLEND);
  glDisable(GL_TEXTURE_2D);
}
```

Código 14: Presenta las gotas de lluvia en pantalla.

2.2.2.3 Explosiones

Este sistema genera todas sus partículas en el momento de su inicialización.

2.2.2.3.1 InitializeParticle

Inicializa las partículas de la explosión ubicándolas en el origen del sistema. Todas las partículas tienen un tamaño constante y su energía varía aleatoriamente.

```
void CExplotion::InitializeParticle(int index)
{
  m_particleList[index].m_pos = m_origin;
  m_particleList[index].m_size = 0.1;
  m_particleList[index].m_velocity.x = FRAND;
  m_particleList[index].m_velocity.y = FRAND;
  m_particleList[index].m_velocity.z = FRAND;
  m_particleList[index].m_energy = 10.0 + m_maxEnergy * FRAND;
}
```

Código 15: Inicializa las partículas de la explosión.

2.2.2.3.2 Update

En este método se aplican las reglas que rigen la animación de la explosión:

a) Primero, se actualiza la posición de las partículas en base a la velocidad y al tiempo transcurrido desde la última actualización.

b) Se disminuye la energía de la partícula.

c) Se verifica el estado de la partícula. Si su energía es menor o igual a cero, o si su posición respecto a Y es menor que cero, la partícula muere.

```
void CExplotion::Update(float elapsedTime)
{
  for (int i = 0; i < m_numParticles; )
  {
    m_particleList[i].m_pos = m_particleList[i].m_pos +
      m_particleList[i].m_velocity * (elapsedTime –
      m_particleList[i].m_startTime);
      m_particleList[i].m_energy -= 0.5f;
    if (m_particleList[i].m_energy <=0 ||
          m_particleList[i].m_pos.y < 0)
    {
      m_particleList[i] = m_particleList[--m_numParticles];
    }
    else
    {
      ++i;
    }
  }
}
```

Código 16: Actualiza las partículas, aplicando las reglas del sistema.

2.2.2.3.3 Render

```
void CExplotion::Render()
{
  glEnable(GL_BLEND);
  glEnable(GL_TEXTURE_2D);
  glBlendFunc(GL_SRC_ALPHA, GL_ONE);
  glBindTexture(GL_TEXTURE_2D, m_texture);
  glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
  CVector partPos;
  float size;
  glBegin(GL_QUADS);
  for (int i = 0; i < m_numParticles; ++i)
  {
      glColor3f(1.0,0.0,0.0);
    partPos = m_particleList[i].m_pos;
    size = m_particleList[i].m_size/2;
       glTexCoord2f(0.0, 1.0);
    glVertex3f(partPos.x - size, partPos.y + size, partPos.z);
    glTexCoord2f(1.0, 1.0);
    glVertex3f(partPos.x + size, partPos.y + size, partPos.z);
    glTexCoord2f(1.0, 0.0);
    glVertex3f(partPos.x + size, partPos.y - size, partPos.z);
    glTexCoord2f(0.0, 0.0);
    glVertex3f(partPos.x - size, partPos.y - size, partPos.z);
  }
  glEnd();
  glDisable(GL_BLEND);
  glDisable(GL_TEXTURE_2D);
}
```

Código 17: Presenta las partículas en pantalla.

2.2.2.4 Fuegos Artificiales

Este sistema genera todas sus partículas en el momento de su inicialización.

2.2.2.4.1 InitializeParticle

Inicializa las partículas de la explosión ubicándolas en el origen del sistema. Todas las partículas tienen un tamaño constante y su energía varía aleatoriamente.

```
void CFireworks::InitializeParticle(int index)
{
  m_particleList[index].m_pos = m_origin;
  m_particleList[index].m_size = 0.1;
  m_particleList[index].m_prevPos = m_origin;
  m_particleList[index].m_velocity.x = FRAND;
  m_particleList[index].m_velocity.y = FRAND;
  m_particleList[index].m_velocity.z = FRAND;
  m_particleList[index].m_energy = 5.0f;
}
```

Código 18: Inicializa las partículas de la explosión.

2.2.2.4.2 Update

En este método se aplican las reglas que rigen la animación de la explosión:

a) Primero, se actualiza la posición de las partículas en base a la velocidad y al tiempo transcurrido desde la última actualización.

b) Se disminuye la energía de la partícula.

c) Se verifica el estado de la partícula. Si su energía es menor o igual a cero, la partícula muere.

```
void CFireworks::Update(float elapsedTime)
{
  for (int i = 0; i < m_numParticles; )
  {
    m_particleList[i].m_pos = m_particleList[i].m_prevPos +
      m_particleList[i].m_velocity * (elapsedTime –
      m_particleList[i].m_startTime);
      m_particleList[i].m_prevPos = m_particleList[i].m_pos;
      m_particleList[i].m_energy -= 0.5f;
    if (m_particleList[i].m_energy <=0)
    {
      m_particleList[i] = m_particleList[--m_numParticles];
    }
    else
    {
      ++i;
    }
  }
}
```

Código 19: Actualiza las partículas, aplicando las reglas del sistema.

2.2.2.4.3 Render

```
void CFireworks::Render()
{
  glEnable(GL_BLEND);
  glEnable(GL_TEXTURE_2D);
  glBlendFunc(GL_SRC_ALPHA, GL_ONE);
  glBindTexture(GL_TEXTURE_2D, m_texture);
  glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
  CVector partPos;
  float size;
  glBegin(GL_QUADS);
  for (int i = 0; i < m_numParticles; ++i)
  {
      glColor4f(FRAND,FRAND,FRAND,1);
    partPos = m_particleList[i].m_pos;
    size = m_particleList[i].m_size/2;
    glTexCoord2f(0.0, 1.0);
    glVertex3f(partPos.x - size, partPos.y + size, partPos.z);
    glTexCoord2f(1.0, 1.0);
    glVertex3f(partPos.x + size, partPos.y + size, partPos.z);
    glTexCoord2f(1.0, 0.0);
    glVertex3f(partPos.x + size, partPos.y - size, partPos.z);
    glTexCoord2f(0.0, 0.0);
    glVertex3f(partPos.x - size, partPos.y - size, partPos.z);
  }
  glEnd();
  glDisable(GL_BLEND);
  glDisable(GL_TEXTURE_2D);
}
```

Código 20: Presenta las partículas en pantalla.

2.2.2.5 Humo

2.2.2.5.1 InitializeParticle

Inicializa las partículas de humo, ubicándolas en el origen del sistema. Todas las partículas tienen un tamaño constante y su energía varía aleatoriamente.

```
void CSmoke::InitializeParticle(int index, float elapsedTime = 0)
{
  m_particleList[index].m_pos = m_origin;
  m_particleList[index].m_size = 0.1;
  m_particleList[index].m_velocity.x = 0.01 * (rand() % 3 - 1);
  m_particleList[index].m_velocity.y = 0.05;
  m_particleList[index].m_velocity.z = 0.01 * (rand() % 3 - 1);
  m_particleList[index].m_energy = 10.0;
  m_particleList[index].m_colorDelta[3] = 0.2 +
    m_maxEnergy * FRAND;
}
```

Código 21: Inicializa las partículas de humo.

2.2.2.5.2 Update

En este método se aplican las reglas que rigen la animación de la explosión:

a) Primero, se actualiza la posición de las partículas en base a la velocidad y al tiempo transcurrido desde la última actualización.

b) Se disminuye la energía de la partícula.

c) Se aumenta la transparencia de la partícula.

d) Se aumenta el tamaño de a partícula.

e) Se verifica el estado de la partícula. Si se ha agotado su energía, o su transparencia es total (1), la partícula muere.

f) Se calculan cuantas partículas deberán emitirse.

g) Se actualiza el tiempo acumulado en base al número de partículas emitidas.

h) Se emiten las nuevas partículas

```
void CSmoke::Update(float elapsedTime)
{
  for (int i = 0; i < m_numParticles; )
  {
    m_particleList[i].m_pos = m_particleList[i].m_pos +
      m_particleList[i].m_velocity * elapsedTime;
       m_particleList[i].m_energy -= 0.01f;
       m_particleList[i].m_colorDelta[3] += 0.1f;
       m_particleList[i].m_size += 0.001;
    if (m_particleList[i].m_energy <=0 ||
          m_particleList[i].m_colorDelta[3] >= 1)
    {
      m_particleList[i] = m_particleList[--m_numParticles];
    }
    else
    {
      ++i;
    }
  }
  m_accumulatedTime += elapsedTime;
  int newParticles = 5 * m_accumulatedTime;
  m_accumulatedTime -= 1.0f/(float)5 * newParticles;
  Emit(newParticles, elapsedTime);}
```

Código 22: Actualiza las partículas, aplicando las reglas del sistema.

2.2.2.5.3 Render

```
void CSmoke::Render()
{
  glEnable(GL_BLEND);
  glEnable(GL_TEXTURE_2D);
  glBlendFunc(GL_SRC_ALPHA, GL_ONE);
  glBindTexture(GL_TEXTURE_2D, m_texture);
  glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
  CVector partPos;
  float size;
  glBegin(GL_QUADS);
  for (int i = 0; i < m_numParticles; ++i)
  {
      glColor4f(0.1,0.1,0.1,m_particleList[i].m_colorDelta[3]);
    partPos = m_particleList[i].m_pos;
    size = m_particleList[i].m_size / 2;
    glTexCoord2f(0.0, 1.0);
    glVertex3f(partPos.x - size, partPos.y + size, partPos.z);
    glTexCoord2f(1.0, 1.0);
    glVertex3f(partPos.x + size, partPos.y + size, partPos.z);
    glTexCoord2f(1.0, 0.0);
    glVertex3f(partPos.x + size, partPos.y - size, partPos.z);
    glTexCoord2f(0.0, 0.0);
    glVertex3f(partPos.x - size, partPos.y - size, partPos.z);
  }
  glEnd();
  glDisable(GL_BLEND);
  glDisable(GL_TEXTURE_2D);
}
```

Código 23: Presenta las partículas en pantalla.

3. MANEJADOR DE SISTEMAS DE PARTICULAS

3.1 Modificaciones al código

Los sistemas generados en la sección anterior muestran la forma de implementación genérica de los mismos, pero no funcionan de forma óptima ni facilitan su reutilización. Por ello se presentan los siguientes cambios y adiciones al código:

3.1.1 Árbol de elementos

Con el objeto de automatizar la animación de los sistemas en el manejador, será necesario generar un árbol de elementos. Para ello se creará una clase nodo y una clase objeto, la cual generará listas doblemente enlazadas de varios niveles.

3.1.2 Manejo de Memoria

El código actual genera el número máximo de partículas que el sistema puede utilizar, aún cuando no se muestran todas en un frame; por lo que hay un desperdicio en utilización de memoria, pues se manejan partículas adicionales que no se utilizan hasta cierto momento.

Para evitar este problema se recodificará el código de las partículas, de tal forma que éstas sean una clase y no una estructura, y se utilizará una lista de partículas que servirá de “balde”, conteniendo el número máximo de éstas que el animador desee en un tiempo determinado. De esta manera, las partículas serán reutilizadas, y no se podrá exceder el número que posea el balde.

3.1.3 Ubicación relativa de las partículas

Si se desea facilitar la reutilización de los sistemas, es necesario que la posición de las partículas sea relativa al emisor, y no absoluta al mundo en el que se encuentran, pues comúnmente se utiliza un sistema en varios lugares simultáneamente, para lo cual se mueve el emisor, y las partículas se presentan en pantalla en posición relativa al emisor.

3.1.4 Utilización de texturas

En el transcurso del desarrollo de esta investigación, se presentó un problema de versiones de compiladores que afecta el desarrollo del código multiplataforma, ya que inhabilita ciertas librerías de sistema que son utilizadas para cargas el archivo BMP de las texturas.

Debido a esto, las texturas serán generadas como un arreglo de caracteres que definirán la textura a ser utilizada.

3.2 Implementación

3.2.1 Árbol de elementos

El árbol de elementos es una lista doblemente enlazada de varios niveles, que permite formar una jerarquía de elementos para facilitar la creación y animación y rendering de las escenas. Se comienza por el código básico de un nodo de la lista.

3.2.1.1 Nodo

Contiene los atributos básicos y representa la parte fundamental de la lista. Es una clase con punteros hacia clases de su mismo tipo, que representan:

a) Padre (parentNode): puntero al nodo de nivel superior del cual procede.

b) Hijo (childNode): puntero al primer nodo de nivel inferior.

c) Anterior (prevNode): puntero al nodo anterior.

d) Siguiente (nextNode): puntero al nodo siguiente.

Los métodos que presenta la clase son:

a) HasParent: Permite ver si el nodo pertenece a un nodo de nivel superior.

b) HasChild: Muestra si el nodo posee nodos en un nivel inferior.

c) IsFirstChild: Muestra si es el primer nodo del nivel en el que se encuentra.

d) IsLastChild: Permite ver si el nodo es el último de la lista en el nivel en el que se encuentra.

e) AttachTo: Agrega el nodo a la lista bajo el nodo indicado como parámetro.

f) Attach: Agrega el nodo parámetro como hijo del nodo actual.

g) Detach: Libera al nodo de la lista.

h) CountNodes: Cuenta el número de nodos bajo él, más uno.

```
class CNode
{
      public:
            CNode *parentNode;
            CNode *childNode;
            CNode *prevNode;
            CNode *nextNode;

            bool HasParent();
            bool HasChild();
            bool IsFirstChild();
            bool IsLastChild();
            void AttachTo(CNode *NewParent);
            void Attach(CNode *newChild);
            void Detach();
            int CountNodes();
            CNode();
            CNode(CNode *Node);
            virtual ~CNode();

};
```

Código 24: Clase CNode.

3.2. Partículas

```
class CParticle : public CNode
{
public:
  CVector  m_pos;
  CVector  m_prevPos;
  CVector  m_velocity;
  CVector  m_acceleration;
  int m_startTime;
  float   m_energy;
  float   m_size;
  float   m_sizeDelta;
  float   m_weight;
  float   m_weightDelta;
  float   m_color[4];
  float   m_colorDelta[4];
};
```

Código 25: Clase CParticle.

3.3 Balde de partículas

```
class CBucket : public CObject
{
 public:
     CBucket(int maxParticles);
     CParticle* GiveParticle();
     void GetParticle(CParticle *Particle) ;
}
CBucket::CBucket(int maxParticles)
{
 CParticle *tempParticle;
  while(maxParticles >=1)
  {
    tempParticle = new CParticle;
    if(tempParticle != NULL)
    {
       Attach((CNode *)tempParticle);
    }
     maxParticles--;
  }
}

CParticle * CBucket::GiveParticle()
{
       return((CParticle *)childNode);
}




void  CBucket::GetParticle(CParticle *Particle)
{
      Particle->Detach();
      Attach((CNode *)Particle);
}
```

Código 26: Clase CBucket.

3.4 Ubicación relativa de las partículas

Esta modificación es sumamente sencilla, sólo es necesario llamar a la matriz identidad para luego movilizar el origen a la posición del emisor, y de ahí dibujar las partículas del sistema. El código adicionado al Render de los sistemas es:

```
glPushMatrix();
    glTranslatef(m_origin.x, m_origin.y, m_origin.z);
   /*…
      Código dee render de cada sistema de partículas
   …*/
glPopMatrix();
```

Código 27: Ubicación relativa de las partículas en relación a su emisor.

3.5 Utilización de texturas

Para la generación alternativa de texturas se utilizará un arreglo de caracteres de dieciséis por dieciséis, que contendrá la representación de la textura en forma binaria (1 y 0), representando el 0 transparencia total y el 1 opacidad.

Es de tomar en cuenta que las texturas son interpretadas de manera inversa, por esto el código 28 representa la textura de una gota de lluvia, y se puede ver invertida.

```
static char *drops[] = {
  "0000011111000000",
  "0000111111100000",
  "0000111111100000",
  "0000110111110000",
  "0000110111110000",
  "0000110111110000",
  "0000110111100000",
  "0000111111100000",
  "0000011111100000",
  "0000011111100000",
  "0000011111000000",
  "0000011111000000",
  "0000001111000000",
  "0000001111000000",
  "0000000110000000",
  "0000000110000000",
  "0000000110000000",
};
```

Código 28: textura invertida de gota de lluvia.

Para convertir el arreglo de caracteres a código RGB para ser aplicado como textura, utilizaremos la siguiente función:

```
    loc = (GLubyte*) floorTexture;
    for (t = 0; t < 16; t++) {
      for (s = 0; s < 16; s++) {
        if (drops[t][s] == '1') {
          loc[0] = 0x00;
          loc[1] = 0x00;
          loc[2] = 0x66;
          loc[3] = 0xff;
        } else {
          loc[0] = 0;
          loc[1] = 0;
          loc[2] = 0;
          loc[3] = 0;
        }
        loc += 4;
      }
    }


    glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
```

Código 29: Función para convertir el arreglo de caracteres a RGB.

3.6 Manejador de Sistemas de Partículas

```
class CPsm : public CObject
{public:
  void  Update(float elapsedTime);
  void  Render();
};
void CPsm::Render()
{
  CParticleSystem *tempParticleSystem;
  tempParticleSystem = NULL;
  if (HasChild())
    {
      tempParticleSystem = (CParticleSystem *)childNode;
      do
      {
        tempParticleSystem->Render();
          tempParticleSystem = (CParticleSystem *)tempParticleSystem-
>nextNode;
      }while(!tempParticleSystem->IsLastChild());
    }
}
void CPsm::Update(float elapsedTime)
{
  CParticleSystem *tempParticleSystem;
  tempParticleSystem = NULL;
  if (HasChild())
    {
      tempParticleSystem = (CParticleSystem *)childNode;
      do
      {
        tempParticleSystem->Update(elapsedTime);
        tempParticleSystem =
             (CParticleSystem *)tempParticleSystem->nextNode;
      }while(!tempParticleSystem->IsLastChild());
    }
}
```

Código 30: Clase CPsm (Particle System Manager).

ANEXO I

Observaciones y limitaciones

Es de gran importancia recalcar que el proyecto contenido en estas páginas, pese a ser un framework de desarrollo para sistemas de partículas, fue implementado en una máquina con las siguientes especificaciones:

- Microprocesador Athlon XP 1400 (1.2 Ghz)
- 384 MB de memoria RAM
- Tarjeta de Video GForce 2 MX250 32 MB

El código fué probado en una máquina con un microprocesador de 350 Mhz y 32 MB de memoria RAM y funcionó correctamente, pero presentó problemas de lentitud después de ser ejecutado por más de 3 minutos.

También fue probado en una máquina con un microprocesador Pentium IV 2.8 Ghz con 512 MB de memoria RAM y funcionó correctamente.

Por lo antes expuesto, es necesario probar el código desarrollado en los equipos finales y hacer las modificaciones necesarias para su correcta animación, esto incluirá el número máximo de partículas que cada sistema tendrá, así como cuantas partículas serán emitidas en cada período de actualización.

ANEXO II

Código Fuente Primera Evaluación
(Microsoft Visual C++ 6.0)

1. Archivos de cabecera

1.1 bitmap.h

```
#include <stdlib.h>
#include <windows.h>
#include <stdio.h>

#define BITMAP_ID   0x4D42

unsigned char *LoadBitmapFile(char *filename,
       BITMAPINFOHEADER *bitmapInfoHeader)
{
  FILE *filePtr;
  BITMAPFILEHEADER  bitmapFileHeader;
  unsigned char    *bitmapImage;
  unsigned int      imageIdx = 0;
  unsigned char     tempRGB;

  filePtr = fopen(filename, "rb");
  if (filePtr == NULL)
    return NULL;

  fread(&bitmapFileHeader, sizeof(BITMAPFILEHEADER), 1, filePtr);

  if (bitmapFileHeader.bfType != BITMAP_ID)
  {
    fclose(filePtr);
    return NULL;
  }

  fread(bitmapInfoHeader, sizeof(BITMAPINFOHEADER), 1, filePtr);

  fseek(filePtr, bitmapFileHeader.bfOffBits, SEEK_SET);

  bitmapImage = (unsigned char*)malloc(bitmapInfoHeader->biSizeImage);

  if (!bitmapImage)
  {
    free(bitmapImage);
    fclose(filePtr);
    return NULL;
  }

  fread(bitmapImage, 1, bitmapInfoHeader->biSizeImage, filePtr);

  if (bitmapImage == NULL)
  {
    fclose(filePtr);
    return NULL;
  }

  for (imageIdx = 0; imageIdx < bitmapInfoHeader->biSizeImage;
```

```
        imageIdx+=3)
  {
    tempRGB = bitmapImage[imageIdx];
    bitmapImage[imageIdx] = bitmapImage[imageIdx + 2];
    bitmapImage[imageIdx + 2] = tempRGB;
  }

  fclose(filePtr);
  return bitmapImage;
}

unsigned char *LoadBitmapFileWithAlpha(char *filename,
   BITMAPINFOHEADER *bitmapInfoHeader)
{
  unsigned char *bitmapImage =
            LoadBitmapFile(filename, bitmapInfoHeader);
  unsigned char *bitmapWithAlpha =
       (unsigned char *)malloc(bitmapInfoHeader->biSizeImage * 4 / 3);

  if (bitmapImage == NULL || bitmapWithAlpha == NULL)
    return NULL;

  for (unsigned int src = 0, dst = 0;
       src < bitmapInfoHeader->biSizeImage; src +=3, dst +=4)
  {
    if (bitmapImage[src] == 0 && bitmapImage[src+1] == 0
                             && bitmapImage[src+2] == 0)
      bitmapWithAlpha[dst+3] = 0;
    else
      bitmapWithAlpha[dst+3] = 0xFF;

    bitmapWithAlpha[dst] = bitmapImage[src];
    bitmapWithAlpha[dst+1] = bitmapImage[src+1];
    bitmapWithAlpha[dst+2] = bitmapImage[src+2];
  }

  free(bitmapImage);

  return bitmapWithAlpha;
}
```

1.2 explotion.h

```
#ifndef __EXPLOTION_H_
#define __EXPLOTION_H_

#include "particles.h"

class CExplotion : public CParticleSystem
{
public:
  CExplotion(int maxParticles, CVector origin, float energy,
    float alpha) ;

  void  Update(float elapsedTime);
  void  Render();

  void  InitializeSystem();
  void  KillSystem();

protected:
  float m_maxEnergy;
  float m_minAlpha;
  void    InitializeParticle(int index, float elapsedTime);
  GLuint  m_texture;
};



CExplotion::CExplotion(int numParticles, CVector origin,
    float energy,float alpha)
: m_maxEnergy(energy), m_minAlpha(alpha),
  CParticleSystem(numParticles, origin)
{
}

void CExplotion::InitializeParticle(int index, float elapsedTime = 0)
{
  m_particleList[index].m_pos = m_origin;
  m_particleList[index].m_size = 0.1;

  m_particleList[index].m_velocity.x = FRAND;
  m_particleList[index].m_velocity.y = FRAND;
  m_particleList[index].m_velocity.z = FRAND;
  m_particleList[index].m_startTime = elapsedTime;
  m_particleList[index].m_energy = 10.0 + m_maxEnergy * FRAND;
}


void CExplotion::Update(float elapsedTime)
{
  for (int i = 0; i < m_numParticles; )
  {
    m_particleList[i].m_pos = m_particleList[i].m_pos + m_particleList
[i].m_velocity * (elapsedTime - m_particleList[i].m_startTime);
      m_particleList[i].m_energy -= 0.5f;
    if (m_particleList[i].m_energy <=0 || m_particleList[i].m_pos.y < 0)
    {
      m_particleList[i] = m_particleList[--m_numParticles];
    }
    else
```

```
    {
      ++i;
    }
  }

}


void CExplotion::Render()
{
  glEnable(GL_BLEND);
  glEnable(GL_TEXTURE_2D);

  glBlendFunc(GL_SRC_ALPHA, GL_ONE);

  glBindTexture(GL_TEXTURE_2D, m_texture);
  glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);

  CVector partPos;
  float size;

  glBegin(GL_QUADS);
  for (int i = 0; i < m_numParticles; ++i)
  {
      glColor3f(1.0,0.0,0.0);
    partPos = m_particleList[i].m_pos;
    size = m_particleList[i].m_size/2;
      glTexCoord2f(0.0, 1.0);
    glVertex3f(partPos.x - size, partPos.y + size, partPos.z);
    glTexCoord2f(1.0, 1.0);
    glVertex3f(partPos.x + size, partPos.y + size, partPos.z);
    glTexCoord2f(1.0, 0.0);
    glVertex3f(partPos.x + size, partPos.y - size, partPos.z);
    glTexCoord2f(0.0, 0.0);
    glVertex3f(partPos.x - size, partPos.y - size, partPos.z);
  }
  glEnd();
  glDisable(GL_BLEND);
  glDisable(GL_TEXTURE_2D);
}


void CExplotion::InitializeSystem()
{
  glGenTextures(1, &m_texture);
  glBindTexture(GL_TEXTURE_2D, m_texture);

  BITMAPINFOHEADER bitmapInfoHeader;
  unsigned char *buffer = LoadBitmapFileWithAlpha("explotion.bmp",
          &bitmapInfoHeader);

  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
GL_LINEAR_MIPMAP_NEAREST);
  glTexImage2D(GL_TEXTURE_2D, 0, 4, bitmapInfoHeader.biWidth,
bitmapInfoHeader.biHeight, 0, GL_RGBA, GL_UNSIGNED_BYTE, buffer);
  gluBuild2DMipmaps(GL_TEXTURE_2D, 4, bitmapInfoHeader.biWidth,
bitmapInfoHeader.biHeight, GL_RGBA, GL_UNSIGNED_BYTE, buffer);
```

```
  free(buffer);
  CParticleSystem::InitializeSystem();
      Emit(m_maxParticles);
}

void CExplotion::KillSystem()
{
  if (glIsTexture(m_texture))
  {
    glDeleteTextures(1, &m_texture);
  }

  CParticleSystem::KillSystem();
}

#endif
```

1.3 fireworks.h

```
#ifndef __FIREWORKS_H_
#define __FIREWORKS_H_

#include "particles.h"

class CFireworks : public CParticleSystem
{
public:
  CFireworks(int maxParticles, CVector origin);

  void  Update(float elapsedTime);
  void  Render();

  void  InitializeSystem();
  void  KillSystem();

protected:
  void    InitializeParticle(int index, float elapsedTime);
  GLuint  m_texture;
};


CFireworks::CFireworks(int numParticles, CVector origin)
: CParticleSystem(numParticles, origin)
{
}

void CFireworks::InitializeParticle(int index, float elapsedTime = 0)
{
  m_particleList[index].m_pos = m_origin;
  m_particleList[index].m_size = 0.1;

  m_particleList[index].m_prevPos = m_origin;

  m_particleList[index].m_velocity.x = FRAND;
  m_particleList[index].m_velocity.y = FRAND;
  m_particleList[index].m_velocity.z = FRAND;
  m_particleList[index].m_startTime = elapsedTime;
  m_particleList[index].m_energy = 25.0f;
}

void CFireworks::Update(float elapsedTime)
{
  for (int i = 0; i < m_numParticles; )
  {
    m_particleList[i].m_pos = m_particleList[i].m_prevPos +
m_particleList[i].m_velocity * (elapsedTime - m_particleList[i].
m_startTime);
      m_particleList[i].m_prevPos = m_particleList[i].m_pos;
      m_particleList[i].m_energy -= 0.5f;
    if (m_particleList[i].m_energy <=0)
    {
      m_particleList[i] = m_particleList[--m_numParticles];
    }
    else
    {
      ++i;
    }
```

```
  }

}

void CFireworks::Render()
{
  glEnable(GL_BLEND);
  glEnable(GL_TEXTURE_2D);

  glBlendFunc(GL_SRC_ALPHA, GL_ONE);

  glBindTexture(GL_TEXTURE_2D, m_texture);
  glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);

  CVector partPos;
  float size;

  glBegin(GL_QUADS);
  for (int i = 0; i < m_numParticles; ++i)
  {
      glColor4f(FRAND,FRAND,FRAND,0.5);
    partPos = m_particleList[i].m_pos;
    size = m_particleList[i].m_size/2;
    glTexCoord2f(0.0, 1.0);
    glVertex3f(partPos.x - size, partPos.y + size, partPos.z);
    glTexCoord2f(1.0, 1.0);
    glVertex3f(partPos.x + size, partPos.y + size, partPos.z);
    glTexCoord2f(1.0, 0.0);
    glVertex3f(partPos.x + size, partPos.y - size, partPos.z);
    glTexCoord2f(0.0, 0.0);
    glVertex3f(partPos.x - size, partPos.y - size, partPos.z);
  }
  glEnd();
  glDisable(GL_BLEND);
  glDisable(GL_TEXTURE_2D);
}

void CFireworks::InitializeSystem()
{
  glGenTextures(1, &m_texture);
  glBindTexture(GL_TEXTURE_2D, m_texture);

  BITMAPINFOHEADER bitmapInfoHeader;
  unsigned char *buffer = LoadBitmapFileWithAlpha("fireworks.bmp",
&bitmapInfoHeader);

  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
GL_LINEAR_MIPMAP_NEAREST);
  glTexImage2D(GL_TEXTURE_2D, 0, 4, bitmapInfoHeader.biWidth,
bitmapInfoHeader.biHeight, 0, GL_RGBA, GL_UNSIGNED_BYTE, buffer);
  gluBuild2DMipmaps(GL_TEXTURE_2D, 4, bitmapInfoHeader.biWidth,
bitmapInfoHeader.biHeight, GL_RGBA, GL_UNSIGNED_BYTE, buffer);

  free(buffer);
  CParticleSystem::InitializeSystem();
      Emit(m_maxParticles);
}
```

```
void CFireworks::KillSystem()
{
  if (glIsTexture(m_texture))
  {
    glDeleteTextures(1, &m_texture);
  }

  CParticleSystem::KillSystem();
}

#endif
```

1.4 particles.h

```
#ifndef __PARTICLES_H_INCLUDED__
#define __PARTICLES_H_INCLUDED__


#include "vectorlib.hpp"

class CParticleSystem;

struct particle_t
{
  CVector  m_pos;
  CVector  m_prevPos;
  CVector  m_velocity;
  CVector  m_acceleration;

  int m_startTime;

  float   m_energy;

  float   m_size;
  float   m_sizeDelta;

  float   m_weight;
  float   m_weightDelta;

  float   m_color[4];
  float   m_colorDelta[4];
};


class CParticleSystem : public CObject
{
public:
     CParticleSystem(int maxParticles, CVector origin);

  virtual void  Update(float elapsedTime)     = 0;
  virtual void  Render()                      = 0;

  virtual int   Emit(int numParticles, float elapsedTime);

  virtual void  InitializeSystem();
  virtual void  KillSystem();

protected:
  virtual void  InitializeParticle(int index, float elapsedTime) = 0;
  particle_t *m_particleList;
  int         m_maxParticles;
  int         m_numParticles;
  CVector   m_origin;

  float       m_accumulatedTime;

  CVector   m_force;
};

CParticleSystem::CParticleSystem(int maxParticles, CVector origin)
{
  m_maxParticles = maxParticles;
```

```
  m_origin = origin;
  m_particleList = NULL;
}

int CParticleSystem::Emit(int numParticles, float elapsedTime = 0)
{
  while (numParticles && (m_numParticles < m_maxParticles))
  {
    InitializeParticle(m_numParticles++, elapsedTime);
    --numParticles;
  }
  return numParticles;
}

void CParticleSystem::InitializeSystem()
{
  if (m_particleList)
  {
    delete[] m_particleList;
    m_particleList = NULL;
  }
  m_particleList = new particle_t[m_maxParticles];

  m_numParticles = 0;
  m_accumulatedTime = 0.0f;
}

void CParticleSystem::KillSystem()
{
  if (m_particleList)
  {
    delete[] m_particleList;
    m_particleList = NULL;
  }

  m_numParticles = 0;
}

#endif // __PARTICLES_H_INCLUDED__
```

1.5 rain.h

```
#ifndef __RAIN_H_
#define __RAIN_H_

#include "particles.h"

const CVector DROP_VELOCITY  (0.0f, -5.0f, 0.0f);
//const CVector VELOCITY_VARIATION  (0.2f, 0.5f, 0.2f);
const float     DROP_SIZE      = 0.05f;
const float     DROPS_PER_SEC  = 200;

class CRain : public CParticleSystem
{
public:
  CRain(int maxParticles, CVector origin, float height, float width,
float depth);

  void  Update(float elapsedTime);
  void  Render();

  void  InitializeSystem();
  void  KillSystem();

protected:
  void    InitializeParticle(int index, float elapsedTime);
  float   m_height;
  float   m_width;
  float   m_depth;

  GLuint  m_texture;
};



CRain::CRain(int numParticles, CVector origin, float height, float
width, float depth)
  : m_height(height), m_width(width), m_depth(depth), CParticleSystem
(numParticles, origin)
{
}

void CRain::InitializeParticle(int index, float elapsedTime = 0)
{
  m_particleList[index].m_pos.y = m_height;
  m_particleList[index].m_pos.x = m_origin.x + FRAND * m_width;
  m_particleList[index].m_pos.z = m_origin.z + FRAND * m_depth;

  m_particleList[index].m_size = DROP_SIZE;

  m_particleList[index].m_velocity.x = DROP_VELOCITY.x ;//+ FRAND *
VELOCITY_VARIATION.x;
  m_particleList[index].m_velocity.y = DROP_VELOCITY.y + FRAND * 0.5;
  m_particleList[index].m_velocity.z = DROP_VELOCITY.z ;//+ FRAND *
VELOCITY_VARIATION.z;
  m_particleList[index].m_startTime = elapsedTime;
}


void CRain::Update(float elapsedTime)
```

```
{
  for (int i = 0; i < m_numParticles; )
  {
    m_particleList[i].m_pos = m_particleList[i].m_pos + m_particleList
[i].m_velocity * elapsedTime;
      m_particleList[i].m_startTime = elapsedTime;
    if (m_particleList[i].m_pos.y <= 0)
    {
      m_particleList[i] = m_particleList[--m_numParticles];
    }
    else
    {
      ++i;
    }
  }

  m_accumulatedTime += elapsedTime;

  int newParticles = DROPS_PER_SEC * m_accumulatedTime;

  m_accumulatedTime -= 1.0f/(float)DROPS_PER_SEC * newParticles;

  Emit(newParticles);
}


void CRain::Render()
{
  glEnable(GL_BLEND);
  glEnable(GL_TEXTURE_2D);

  glBlendFunc(GL_SRC_ALPHA, GL_ONE);

  glBindTexture(GL_TEXTURE_2D, m_texture);
  glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);

  CVector partPos;
  float size;

  glBegin(GL_QUADS);
  for (int i = 0; i < m_numParticles; ++i)
  {
      glColor4f(0.0,0.0,1.0,0.6);
    partPos = m_particleList[i].m_pos;
    size = m_particleList[i].m_size;
    glTexCoord2f(0.0, 1.0);
    glVertex3f(partPos.x, partPos.y, partPos.z);
    glTexCoord2f(1.0, 1.0);
    glVertex3f(partPos.x + size, partPos.y, partPos.z);
    glTexCoord2f(1.0, 0.0);
    glVertex3f(partPos.x + size, partPos.y - size, partPos.z);
    glTexCoord2f(0.0, 0.0);
    glVertex3f(partPos.x, partPos.y - size, partPos.z);
  }
  glEnd();


  glDisable(GL_BLEND);
  glDisable(GL_TEXTURE_2D);
}
```

```
void CRain::InitializeSystem()
{
  glGenTextures(1, &m_texture);
  glBindTexture(GL_TEXTURE_2D, m_texture);

  BITMAPINFOHEADER bitmapInfoHeader;
  unsigned char *buffer = LoadBitmapFileWithAlpha("rain.bmp",
&bitmapInfoHeader);

  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
GL_LINEAR_MIPMAP_NEAREST);
  glTexImage2D(GL_TEXTURE_2D, 0, 4, bitmapInfoHeader.biWidth,
bitmapInfoHeader.biHeight, 0, GL_RGBA, GL_UNSIGNED_BYTE, buffer);
  gluBuild2DMipmaps(GL_TEXTURE_2D, 4, bitmapInfoHeader.biWidth,
bitmapInfoHeader.biHeight, GL_RGBA, GL_UNSIGNED_BYTE, buffer);

  free(buffer);

  CParticleSystem::InitializeSystem();
}

void CRain::KillSystem()
{
  if (glIsTexture(m_texture))
  {
    glDeleteTextures(1, &m_texture);
  }

  CParticleSystem::KillSystem();
}

#endif
```

1.6 smoke.h

```
#ifndef __SMOKE_H_
#define __SMOKE_H_

#include "particles.h"

class CSmoke : public CParticleSystem
{
public:
  CSmoke(int maxParticles, CVector origin, float energy, float alpha) ;

  void  Update(float elapsedTime);
  void  Render();

  void  InitializeSystem();
  void  KillSystem();

protected:
  float m_maxEnergy;
  float m_minAlpha;
  void    InitializeParticle(int index, float elapsedTime);
  GLuint  m_texture;
};

CSmoke::CSmoke(int numParticles, CVector origin,  float energy,float
alpha)
: m_maxEnergy(energy), m_minAlpha(alpha), CParticleSystem(numParticles,
origin)
{

}

void CSmoke::InitializeParticle(int index, float elapsedTime = 0)
{
  m_particleList[index].m_pos = m_origin;
  m_particleList[index].m_size = 0.1;

  m_particleList[index].m_velocity.x = 0.01 * (rand() % 3 - 1);
  m_particleList[index].m_velocity.y = 0.05;
  m_particleList[index].m_velocity.z = 0.01 * (rand() % 3 - 1);
  m_particleList[index].m_energy = 10.0;
  m_particleList[index].m_colorDelta[3] = 0.2 + m_maxEnergy * FRAND;;
}

void CSmoke::Update(float elapsedTime)
{
  for (int i = 0; i < m_numParticles; )
  {
    m_particleList[i].m_pos = m_particleList[i].m_pos + m_particleList
[i].m_velocity * elapsedTime;
      m_particleList[i].m_energy -= 0.01f;
      m_particleList[i].m_colorDelta[3] += 0.1f;
      m_particleList[i].m_size += 0.001;

    if (m_particleList[i].m_energy <=0 || m_particleList[i].m_pos.y < 0)
    {
      m_particleList[i] = m_particleList[--m_numParticles];
    }
    else
```

```
    {
      ++i;
    }
  }
  m_accumulatedTime += elapsedTime;

  int newParticles = 5 * m_accumulatedTime;

  m_accumulatedTime -= 1.0f/(float)5 * newParticles;

  Emit(newParticles, elapsedTime);}

void CSmoke::Render()
{
  glEnable(GL_BLEND);
  glEnable(GL_TEXTURE_2D);

  glBlendFunc(GL_SRC_ALPHA, GL_ONE);

  glBindTexture(GL_TEXTURE_2D, m_texture);
  glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);

  CVector partPos;
  float size;

  glBegin(GL_QUADS);
  for (int i = 0; i < m_numParticles; ++i)
  {
    glColor4f(0.1,0.1,0.1,m_particleList[i].m_colorDelta[3]);
    partPos = m_particleList[i].m_pos;
    size = m_particleList[i].m_size / 2;
    glTexCoord2f(0.0, 1.0);
    glVertex3f(partPos.x - size, partPos.y + size, partPos.z);
    glTexCoord2f(1.0, 1.0);
    glVertex3f(partPos.x + size, partPos.y + size, partPos.z);
    glTexCoord2f(1.0, 0.0);
    glVertex3f(partPos.x + size, partPos.y - size, partPos.z);
    glTexCoord2f(0.0, 0.0);
    glVertex3f(partPos.x - size, partPos.y - size, partPos.z);
  }
  glEnd();
  glDisable(GL_BLEND);
  glDisable(GL_TEXTURE_2D);
}


void CSmoke::InitializeSystem()
{
  glGenTextures(1, &m_texture);
  glBindTexture(GL_TEXTURE_2D, m_texture);

  BITMAPINFOHEADER bitmapInfoHeader;
  unsigned char *buffer = LoadBitmapFileWithAlpha("smoke.bmp",
&bitmapInfoHeader);

  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
GL_LINEAR_MIPMAP_NEAREST);
```

```
  glTexImage2D(GL_TEXTURE_2D, 0, 4, bitmapInfoHeader.biWidth,
bitmapInfoHeader.biHeight, 0, GL_RGBA, GL_UNSIGNED_BYTE, buffer);
  gluBuild2DMipmaps(GL_TEXTURE_2D, 4, bitmapInfoHeader.biWidth,
bitmapInfoHeader.biHeight, GL_RGBA, GL_UNSIGNED_BYTE, buffer);

  free(buffer);
  CParticleSystem::InitializeSystem();

}

void CSmoke::KillSystem()
{
  if (glIsTexture(m_texture))
  {
    glDeleteTextures(1, &m_texture);
  }

  CParticleSystem::KillSystem();
}

#endif
```

1.7 snow.h

```
#ifndef __SNOW_H_
#define __SNOW_H_

#include "particles.h"
#include "bitmap.h"

const CVector SNOWFLAKE_VELOCITY  (0.0f, -0.1f, 0.0f);
const CVector VELOCITY_VARIATION  (0.2f, 0.5f, 0.2f);
const float     SNOWFLAKE_SIZE      = 0.02f;
const float     SNOWFLAKES_PER_SEC  = 200;

class CSnowstorm : public CParticleSystem
{
public:
  CSnowstorm(int maxParticles, CVector origin, float height, float
width, float depth);

  void  Update(float elapsedTime);
  void  Render();

  void  InitializeSystem();
  void  KillSystem();

protected:
  void    InitializeParticle(int index, float elapsedTime);
  float   m_height;
  float   m_width;
  float   m_depth;

  GLuint  m_texture;
};

CSnowstorm::CSnowstorm(int numParticles, CVector origin, float height,
float width, float depth)
  : m_height(height), m_width(width), m_depth(depth), CParticleSystem
(numParticles, origin)
{
}

void CSnowstorm::InitializeParticle(int index, float elapsedTime = 0)
{
  m_particleList[index].m_pos.y = m_height;
  m_particleList[index].m_pos.x = m_origin.x + FRAND * m_width;
  m_particleList[index].m_pos.z = m_origin.z + FRAND * m_depth;

  m_particleList[index].m_size = SNOWFLAKE_SIZE;

  m_particleList[index].m_velocity.x = SNOWFLAKE_VELOCITY.x + FRAND *
VELOCITY_VARIATION.x;
  m_particleList[index].m_velocity.y = SNOWFLAKE_VELOCITY.y + FRAND *
VELOCITY_VARIATION.y;
  m_particleList[index].m_velocity.z = SNOWFLAKE_VELOCITY.z + FRAND *
VELOCITY_VARIATION.z;
  m_particleList[index].m_startTime = elapsedTime;
}


void CSnowstorm::Update(float elapsedTime)
```

```
{

  for (int i = 0; i < m_numParticles; )
  {
    m_particleList[i].m_pos = m_particleList[i].m_pos + m_particleList
[i].m_velocity * (elapsedTime - m_particleList[i].m_startTime);

    if (m_particleList[i].m_pos.y <= m_origin.y)
    {
      m_particleList[i] = m_particleList[--m_numParticles];
    }
    else
    {
      ++i;
    }
  }

  m_accumulatedTime += elapsedTime;

  int newParticles = SNOWFLAKES_PER_SEC * m_accumulatedTime;

  m_accumulatedTime -= 1.0f/(float)SNOWFLAKES_PER_SEC * newParticles;

  Emit(newParticles, elapsedTime);
}


void CSnowstorm::Render()
{
  glEnable(GL_BLEND);
  glEnable(GL_TEXTURE_2D);

  glBlendFunc(GL_SRC_ALPHA, GL_ONE);

  glBindTexture(GL_TEXTURE_2D, m_texture);
  glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);

  CVector partPos;
  float size;

  glBegin(GL_QUADS);
  for (int i = 0; i < m_numParticles; ++i)
  {
      glColor3f(1.0,1.0,1.0);
    partPos = m_particleList[i].m_pos;
    size = m_particleList[i].m_size;
    glTexCoord2f(0.0, 1.0);
    glVertex3f(partPos.x, partPos.y, partPos.z);
    glTexCoord2f(1.0, 1.0);
    glVertex3f(partPos.x + size, partPos.y, partPos.z);
    glTexCoord2f(1.0, 0.0);
    glVertex3f(partPos.x + size, partPos.y - size, partPos.z);
    glTexCoord2f(0.0, 0.0);
    glVertex3f(partPos.x, partPos.y - size, partPos.z);
  }
  glEnd();


  glDisable(GL_BLEND);
  glDisable(GL_TEXTURE_2D);
}
```

```
void CSnowstorm::InitializeSystem()
{
  glGenTextures(1, &m_texture);
  glBindTexture(GL_TEXTURE_2D, m_texture);

 BITMAPINFOHEADER bitmapInfoHeader;
 unsigned char *buffer = LoadBitmapFileWithAlpha("snowstorm.bmp",
&bitmapInfoHeader);

  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
GL_LINEAR_MIPMAP_NEAREST);
  glTexImage2D(GL_TEXTURE_2D, 0, 4, bitmapInfoHeader.biWidth,
bitmapInfoHeader.biHeight, 0, GL_RGBA, GL_UNSIGNED_BYTE, buffer);
  gluBuild2DMipmaps(GL_TEXTURE_2D, 4, bitmapInfoHeader.biWidth,
bitmapInfoHeader.biHeight, GL_RGBA, GL_UNSIGNED_BYTE, buffer);

  free(buffer);

  CParticleSystem::InitializeSystem();
}

void CSnowstorm::KillSystem()
{
  if (glIsTexture(m_texture))
  {
    glDeleteTextures(1, &m_texture);
  }

  CParticleSystem::KillSystem();
}

#endif
```

1.8 vectorlib.hpp

```
#ifndef __VECTOR_H
#define __VECTOR_H

#include <math.h>
#include <stdlib.h>

/*
     VECTOR.H

     CVector class

     OpenGL Game Programming
     by Kevin Hawkins and Dave Astle

     Some operators of the CVector class based on
     operators of the CVector class by Bas Kuenen.
     Copyright (c) 2000 Bas Kuenen. All Rights Reserved.
     homepage: baskuenen.cfxweb.net
*/

#define PI         (3.14159265359f)
#define DEG2RAD(a)      (PI/180*(a))
#define RAD2DEG(a)      (180/PI*(a))
#define FRAND   (((float)rand()-(float)rand())/RAND_MAX)
#define Clamp(x, min, max)  x = (x<min  ? min : x<max ? x : max);

#define SQUARE(x)  (x)*(x)


typedef float scalar_t;

class CVector
{
public:
     scalar_t x;
     scalar_t y;
     scalar_t z;    // x,y,z coordinates

public:
     CVector(scalar_t a = 0, scalar_t b = 0, scalar_t c = 0) : x(a), y
(b), z(c) {}
     CVector(const CVector &vec) : x(vec.x), y(vec.y), z(vec.z) {}

      // vector index
      scalar_t &operator[](const long idx)
      {
            return *((&x)+idx);
      }

     // vector assignment
     const CVector &operator=(const CVector &vec)
     {
          x = vec.x;
          y = vec.y;
          z = vec.z;

          return *this;
     }
```

```
// vecector equality
const bool operator==(const CVector &vec) const
{
     return ((x == vec.x) && (y == vec.y) && (z == vec.z));
}

// vecector inequality
const bool operator!=(const CVector &vec) const
{
     return !(*this == vec);
}

// vector add
const CVector operator+(const CVector &vec) const
{
     return CVector(x + vec.x, y + vec.y, z + vec.z);
}

// vector add (opposite of negation)
const CVector operator+() const
{
     return CVector(*this);
}

// vector increment
const CVector& operator+=(const CVector& vec)
{    x += vec.x;
     y += vec.y;
     z += vec.z;
     return *this;
}

// vector subtraction
const CVector operator-(const CVector& vec) const
{
     return CVector(x - vec.x, y - vec.y, z - vec.z);
}

// vector negation
const CVector operator-() const
{
     return CVector(-x, -y, -z);
}

// vector decrement
const CVector &operator-=(const CVector& vec)
{
     x -= vec.x;
     y -= vec.y;
     z -= vec.z;

     return *this;
}

// scalar self-multiply
const CVector &operator*=(const scalar_t &s)
{
     x *= s;
     y *= s;
     z *= s;
```

```
          return *this;
     }

     // scalar self-divecide
     const CVector &operator/=(const scalar_t &s)
     {
          const float recip = 1/s; // for speed, one divecision

          x *= recip;
          y *= recip;
          z *= recip;

          return *this;
     }

     // post multiply by scalar
     const CVector operator*(const scalar_t &s) const
     {
          return CVector(x*s, y*s, z*s);
     }

     // pre multiply by scalar
     friend inline const CVector operator*(const scalar_t &s, const
CVector &vec)
     {
          return vec*s;
     }

      const CVector operator*(const CVector& vec) const
      {
            return CVector(x*vec.x, y*vec.y, z*vec.z);
      }

      // post multiply by scalar
     /*friend inline const CVector operator*(const CVector &vec, const
scalar_t &s)
     {
          return CVector(vec.x*s, vec.y*s, vec.z*s);
     }*/

    // divide by scalar
     const CVector operator/(scalar_t s) const
     {
          s = 1/s;

          return CVector(s*x, s*y, s*z);
     }

     // cross product
     const CVector CrossProduct(const CVector &vec) const
     {
          return CVector(y*vec.z - z*vec.y, z*vec.x - x*vec.z, x*vec.y
- y*vec.x);
     }

     // cross product
     const CVector operator^(const CVector &vec) const
     {
          return CVector(y*vec.z - z*vec.y, z*vec.x - x*vec.z, x*vec.y
- y*vec.x);
```

```
}

// dot product
const scalar_t DotProduct(const CVector &vec) const
{
     return x*vec.x + y*vec.x + z*vec.z;
}

// dot product
const scalar_t operator%(const CVector &vec) const
{
     return x*vec.x + y*vec.x + z*vec.z;
}

// length of vector
const scalar_t Length() const
{
     return (scalar_t)sqrt((double)(x*x + y*y + z*z));
}

// return the unit vector
const CVector UnitVector() const
{
     return (*this) / Length();
}

// normalize this vector
void Normalize()
{
     (*this) /= Length();
}

const scalar_t operator!() const
{
     return sqrtf(x*x + y*y + z*z);
}

// return vector with specified length
const CVector operator | (const scalar_t length) const
{
     return *this * (length / !(*this));
}

// set length of vector equal to length
const CVector& operator |= (const float length)
{
     return *this = *this | length;
}

// return angle between two vectors
const float inline Angle(const CVector& normal) const
{
     return acosf(*this % normal);
}

// reflect this vector off surface with normal vector
const CVector inline Reflection(const CVector& normal) const
{
     const CVector vec(*this | 1);     // normalize this vector
     return (vec - normal * 2.0 * (vec % normal)) * !*this;
}
```

```
	// rotate angle degrees about a normal
	const CVector inline Rotate(const float angle, const CVector&
normal) const
	{
		const float cosine = cosf(angle);
		const float sine = sinf(angle);

		return CVector(*this * cosine + ((normal * *this) * (1.0f -
cosine)) *
				normal + (*this ^ normal) * sine);
	}
};

#endif
```

2. Archivos de implementación

2.1 main.cpp

```
#include <iostream.h>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <gl\glut.h>

#include <math.h>

const float pi = 3.14f;
void init(void)
{
glClearColor(0.0f, 0.0f, 0.0f, 0.0f) ; //set background colour to black
glShadeModel(GL_SMOOTH) ; //use flat shading for polygons/faces
}
void display(void)
{
int i,start=0;
float x,y;
const float max = 10;
glClear(GL_COLOR_BUFFER_BIT) ; //clear colour buffer before drawing
for(start=0; start<=1000; start++)
{
const float blue = (float)start/1000; //calculate value for blue
glBegin(GL_LINE_STRIP); //draw joined up lines
for(i=0; i<=max; i++)
{
const float red = (float)i/max; //calculate value for red
x = i * sin((start+i)*2*pi/max)/max;
y = i * cos((start+i)*2*pi/max)/max;
glColor3f(red, 0.0f, blue); //line colour
glVertex3f(x, y, 0.0f); //draw line to a vertex
}
glEnd();
glutSwapBuffers(); //swap drawing from memory to video memory
}
}
int main(int argc, char** argv)
{
glutInit (&argc, argv) ;
glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB) ; //use double buffering
to draw to screen
glutInitWindowSize (500, 500) ; //set size of drawing window (in pixels)
glutInitWindowPosition (100, 100) ; //set start up position, relative
to top left corner of screen
glutCreateWindow ("Tutorial 2: A simple drawing application using
OpenGL") ; //title
init() ; //call function to configure initial OpenGL settings
glutDisplayFunc (display) ;
glutMainLoop () ;
return 0 ;
}
```

ANEXO III

Código Fuente Segunda Evaluación (Bloodshed Dev-C++ 4.0)

1. Archivos de encabezado

1.1 bucket.h

```
#ifndef __BUCKET_H_INCLUDED__
#define __BUCKET_H_INCLUDED__
#include "Particles.h"

class CBucket : public CObject
{
 public:
     CBucket(int maxParticles);
     CParticle* GiveParticle();
     void GetParticle(CParticle *Particle) ;
};

#endif // __BUCKET_H_INCLUDED__
```

1.2 explotion.h

```
#ifndef __EXPLOTION_H_
#define __EXPLOTION_H_

#include "particles.h"

class CExplotion : public CParticleSystem
{
public:
  CExplotion(int maxParticles, CVector origin, CObject *MyBucket, float
energy, float alpha) ;

  void  Update(float elapsedTime);
  void  Render();

  void  InitializeSystem();
  void  KillSystem();

protected:
  float m_maxEnergy;
  float m_minAlpha;
  void  InitializeParticle(CParticle *particleToInitialize);
  void  KillParticles();
  GLuint  m_texture;
};




CExplotion::CExplotion(int   numParticles,   CVector   origin,   CObject
*MyBucket, float energy,float alpha)
: m_maxEnergy(energy), m_minAlpha(alpha), CParticleSystem(numParticles,
origin, MyBucket)
{
}

void CExplotion::InitializeParticle(CParticle *particleToInitialize)
{
  particleToInitialize->m_pos = m_origin;
  particleToInitialize->m_size = 0.1;
  particleToInitialize->m_velocity.x = FRAND;
  particleToInitialize->m_velocity.y = FRAND;
  particleToInitialize->m_velocity.z = FRAND;
  particleToInitialize->m_energy = 10.0 + m_maxEnergy * FRAND;

  Attach(particleToInitialize);
}


void CExplotion::Update(float elapsedTime)
{
  CParticle *tempParticle;
  if(HasChild())
  {
    tempParticle = (CParticle *)childNode;
   for(int iLoop = 1; iLoop <= CountNodes() -1; iLoop++)
    {
      tempParticle->m_pos = tempParticle->m_pos +
```

```
        (tempParticle->m_velocity * elapsedTime);
      tempParticle->m_energy -= 0.5f;
      tempParticle = (CParticle *)tempParticle->nextNode;
    }
  }
  KillParticles();
}

void CExplotion::KillParticles()
{
   if(HasChild())
   {
     CParticle *tempParticle, *tempParticle1;
     tempParticle = NULL;
     tempParticle = (CParticle *)childNode;
     while(tempParticle!=NULL)
     {
       if ((tempParticle->m_energy <= 0) ||
           (tempParticle->m_pos.y <= 0.0))
       {
         ((CBucket *)m_ParticleBucket)->GetParticle(tempParticle);
          tempParticle = (CParticle *)childNode;
          m_numParticles--;
       }
       else
       {
         tempParticle = (CParticle *)tempParticle->nextNode;
         if(tempParticle->IsLastChild())
         {
           tempParticle = NULL;
         }
       }
     }
   }
}

void CExplotion::Render()
{
  CParticle *tempParticle;
  glEnable(GL_BLEND);
  glEnable(GL_TEXTURE_2D);

  glBlendFunc(GL_SRC_ALPHA, GL_ONE);

  glBindTexture(GL_TEXTURE_2D, m_texture);
  glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);

  CVector partPos;
  float size;
  glPushMatrix();
  glTranslatef(m_origin.x, m_origin.y, m_origin.z);
  glBegin(GL_QUADS);
  if(HasChild())
  {
    tempParticle = (CParticle *)childNode;
    for(int iLoop = 1; iLoop <= CountNodes() -1; iLoop++)
    {
      partPos = tempParticle->m_pos;
        glColor3f(1.0,0.0,0.0);
      size = tempParticle->m_size / 2;
      glTexCoord2f(0.0, 1.0);
```

```
      glVertex3f(partPos.x - size, partPos.y + size, partPos.z);
      glTexCoord2f(1.0, 1.0);
      glVertex3f(partPos.x + size, partPos.y + size, partPos.z);
      glTexCoord2f(1.0, 0.0);
      glVertex3f(partPos.x + size, partPos.y - size, partPos.z);
      glTexCoord2f(0.0, 0.0);
      glVertex3f(partPos.x - size, partPos.y - size, partPos.z);
      tempParticle = (CParticle *)tempParticle->nextNode;
    }
   }
  glEnd();
  glPopMatrix();
  glDisable(GL_BLEND);
  glDisable(GL_TEXTURE_2D);
}


void CExplotion::InitializeSystem()
{
 glGenTextures(1, &m_texture);
  glBindTexture(GL_TEXTURE_2D, m_texture);
  GLubyte explosionTexture[16][16][4];
  GLubyte *loc;
  int s, t;

 static char *explosion[] = {
  "1111111111111111",
  "1111011111011111",
  "1111011111011111",
  "1111101010111111",
  "1111101010111111",
  "1111110001011111",
  "1111010000110001",
  "1111100000000111",
  "1000000000111111",
  "1111000000001111",
  "1111010000111111",
  "1111101000011111",
  "1111010000101111",
  "1110111101110111",
  "1111111101111011",
  "1111111101111111",
};

  /* Setup RGB image for the texture. */
  loc = (GLubyte*) explosionTexture;
  for (t = 0; t < 16; t++) {
    for (s = 0; s < 16; s++) {
      if (explosion[t][s] == '0') {
        loc[0] = 0xFF;
        loc[1] = 0xFF;
        loc[2] = 0xFF;
        loc[3] = 0xFF;
      } else {
        loc[0] = 0;
        loc[1] = 0;
        loc[2] = 0;
        loc[3] = 0;
      }
      loc += 4;
    }
```

```
  }

  glPixelStorei(GL_UNPACK_ALIGNMENT, 1);

  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
            glTexParameteri(GL_TEXTURE_2D,       GL_TEXTURE_MIN_FILTER,
GL_LINEAR_MIPMAP_NEAREST);
      glTexImage2D(GL_TEXTURE_2D,   0,   4,   16,   16,   0,   GL_RGBA,
GL_UNSIGNED_BYTE, explosionTexture);
       gluBuild2DMipmaps(GL_TEXTURE_2D,    4,    16,    16,    GL_RGBA,
GL_UNSIGNED_BYTE, explosionTexture);

  CParticleSystem::InitializeSystem();
  Emit(m_maxParticles);
}

void CExplotion::KillSystem()
{
  if (glIsTexture(m_texture))
  {
    glDeleteTextures(1, &m_texture);
  }

  CParticleSystem::KillSystem();
}

#endif
```

1.3 fireworks.h

```
#ifndef __FIREWORKS_H_
#define __FIREWORKS_H_

#include "particles.h"

class CFireworks : public CParticleSystem
{
public:
  CFireworks(int maxParticles, CVector origin, CObject *MyBucket);

  void  Update(float elapsedTime);
  void  Render();

  void  InitializeSystem();
  void  KillSystem();

protected:
  void  InitializeParticle(CParticle *particleToInitialize);
  void  KillParticles();
  GLuint  m_texture;
};

CFireworks::CFireworks(int   numParticles,   CVector   origin,   CObject
*MyBucket)
: CParticleSystem(numParticles, origin, MyBucket)
{
}

void CFireworks::InitializeParticle(CParticle *particleToInitialize)
{
  particleToInitialize->m_pos = m_origin;
  particleToInitialize->m_size = 0.1;

  particleToInitialize->m_prevPos = m_origin;

  particleToInitialize->m_velocity.x = FRAND;
  particleToInitialize->m_velocity.y = FRAND;
  particleToInitialize->m_velocity.z = FRAND;
  particleToInitialize->m_energy = 5.0f;

  Attach(particleToInitialize);
}


void CFireworks::Update(float elapsedTime)
{
  CParticle *tempParticle;
  if(HasChild())
  {
    tempParticle = (CParticle *)childNode;
   for(int iLoop = 1; iLoop <= CountNodes() -1; iLoop++)
    {
      tempParticle->m_pos = tempParticle->m_pos +
          (tempParticle->m_velocity * elapsedTime);
      tempParticle->m_energy -= 0.5f;
      tempParticle->m_prevPos = tempParticle->m_pos;
      tempParticle = (CParticle *)tempParticle->nextNode;
    }
```

```
  }
  KillParticles();
}

void CFireworks::KillParticles()
{
   if(HasChild())
   {
     CParticle *tempParticle, *tempParticle1;
     tempParticle = NULL;
     tempParticle = (CParticle *)childNode;
     while(tempParticle!=NULL)
     {
       if ((tempParticle->m_energy <= 0))
       {
         ((CBucket *)m_ParticleBucket)->GetParticle(tempParticle);
          tempParticle = (CParticle *)childNode;
          m_numParticles--;
       }
       else
       {
         tempParticle = (CParticle *)tempParticle->nextNode;
         if(tempParticle->IsLastChild())
         {
           tempParticle = NULL;
         }
       }
     }
   }
}

void CFireworks::Render()
{
  CParticle *tempParticle;
  glEnable(GL_BLEND);
  glEnable(GL_TEXTURE_2D);

  glBlendFunc(GL_SRC_ALPHA, GL_ONE);

  glBindTexture(GL_TEXTURE_2D, m_texture);
  glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);

  CVector partPos;
  float size;
  glPushMatrix();
  glTranslatef(m_origin.x, m_origin.y, m_origin.z);
  glBegin(GL_QUADS);
  if(HasChild())
  {
    tempParticle = (CParticle *)childNode;
    for(int iLoop = 1; iLoop <= CountNodes() -1; iLoop++)
    {
      partPos = tempParticle->m_pos;
      glColor4f(FRAND,FRAND,FRAND,1);
      size = tempParticle->m_size / 2;
      glTexCoord2f(0.0, 1.0);
      glVertex3f(partPos.x - size, partPos.y + size, partPos.z);
      glTexCoord2f(1.0, 1.0);
      glVertex3f(partPos.x + size, partPos.y + size, partPos.z);
      glTexCoord2f(1.0, 0.0);
      glVertex3f(partPos.x + size, partPos.y - size, partPos.z);
```

```
      glTexCoord2f(0.0, 0.0);
      glVertex3f(partPos.x - size, partPos.y - size, partPos.z);
      tempParticle = (CParticle *)tempParticle->nextNode;
    }
   }
  glEnd();
  glPopMatrix();
  glDisable(GL_BLEND);
  glDisable(GL_TEXTURE_2D);
}

void CFireworks::InitializeSystem()
{
  glGenTextures(1, &m_texture);
  glBindTexture(GL_TEXTURE_2D, m_texture);
  GLubyte fireworksTexture[16][16][4];
  GLubyte *loc;
  int s, t;

 static char *fireworks[] = {
  "1111111111111111",
  "1111011111011111",
  "1111011111011111",
  "1111101010111111",
  "1111101010111111",
  "1111110001011111",
  "1111010000110001",
  "1111100000000111",
  "1000000000111111",
  "1111000000001111",
  "1111010000111111",
  "1111101000011111",
  "1111010000101111",
  "1110111101110111",
  "1111111101111011",
  "1111111101111111",
};

  /* Setup RGB image for the texture. */
  loc = (GLubyte*) fireworksTexture;
  for (t = 0; t < 16; t++) {
    for (s = 0; s < 16; s++) {
      if (fireworks[t][s] == '0') {
        loc[0] = 0xFF;
        loc[1] = 0xFF;
        loc[2] = 0xFF;
        loc[3] = 0xFF;
      } else {
        loc[0] = 0;
        loc[1] = 0;
        loc[2] = 0;
        loc[3] = 0;
      }
      loc += 4;
    }
  }

  glPixelStorei(GL_UNPACK_ALIGNMENT, 1);

  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
```

```
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                  GL_LINEAR_MIPMAP_NEAREST);
  glTexImage2D(GL_TEXTURE_2D, 0, 4, 16, 16, 0, GL_RGBA,
                GL_UNSIGNED_BYTE, fireworksTexture);
  gluBuild2DMipmaps(GL_TEXTURE_2D, 4, 16, 16, GL_RGBA,
                    GL_UNSIGNED_BYTE, fireworksTexture);

  CParticleSystem::InitializeSystem();
      Emit(m_maxParticles);
}

void CFireworks::KillSystem()
{
  if (glIsTexture(m_texture))
  {
    glDeleteTextures(1, &m_texture);
  }

  CParticleSystem::KillSystem();
}

#endif
```

1.4 Objeto.h

```
#ifndef _OBJETO_H_
#define _OBJETO_H_

#include "general.hpp"
#include "gl/glut.h"

class CObject : public CNode {
protected:
	bool bDelete;

	virtual void OnDraw() {}
	virtual void OnCollision() {}
	virtual void OnPrepare(){}
	virtual void OnAnimate(scalar_t deltaTime);

public:
	CVector position;
	CVector velocity;
	CVector acceleration;
	scalar_t size;

	CObject::CObject() { bDelete = false; }
	CObject::~CObject() {}

	virtual void Load() {}
	virtual void Unload() {}

	void Draw() ;
	void Animate(scalar_t deltaTime);
	void ProcessCollisions() {}
	void Prepare();
	CObject *FindRoot()
	{
		if (parentNode)
			return ((CObject*)parentNode)->FindRoot();

		return this;
	}
};

#endif
```

1.5 particles.h

```
#ifndef __PARTICLES_H_INCLUDED__
#define __PARTICLES_H_INCLUDED__

#include "vectorlib.hpp"
#include "Objeto.h"

class CParticle : public CNode
{
public:
  CVector  m_pos;
  CVector  m_prevPos;
  CVector  m_velocity;
  CVector  m_acceleration;
  int m_startTime;
  float   m_energy;
  float   m_size;
  float   m_sizeDelta;
  float   m_weight;
  float   m_weightDelta;
  float   m_color[4];
  float   m_colorDelta[4];
};


class CParticleSystem : public CObject
{
 public:
     CParticleSystem(int maxParticles, CVector origin, CObject
*MyBucket);
     virtual void  Update(float elapsedTime) =0;
     virtual void  Render() =0;
     int   Emit(int numParticles);
     void  InitializeSystem();
     void  KillSystem();
     void  MoveEmmit(CVector origin);
     void  MoveOrigin(CVector origin);
  protected:
     virtual void  InitializeParticle(CParticle *particleToInitialize)
= 0;
     CObject    *m_ParticleBucket;
     int        m_maxParticles;
     int        m_numParticles;
     CVector    m_origin;
     float      m_accumulatedTime;
     CVector    m_force;
};



#endif // __PARTICLES_H_INCLUDED__
```

1.6 psm.h

```
#ifndef __PSM_H_INCLUDED__
#define __PSM_H_INCLUDED__
#include "Particles.h"

class CPsm : public CObject
{
public:
  void  Update(float elapsedTime);
  void  Render();
};

#endif // __PSM_H_INCLUDED__
```

1.7 rain.h

```
#ifndef __RAIN_H_
#define __RAIN_H_

#include "particles.h"

const CVector DROP_VELOCITY  (0.0f, -5.0f, 0.0f);
const float     DROP_SIZE      = 0.05f;
const float     DROPS_PER_SEC  = 200;

class CRain : public CParticleSystem
{
public:
  CRain(int maxParticles, CVector origin, CObject *MyBucket,
         float height, float width, float depth);
  void  Update(float elapsedTime);
  void  Render();
  void  InitializeSystem();
  void  KillSystem();

protected:
  void  InitializeParticle(CParticle *particleToInitialize);
  float   m_height;
  float   m_width;
  float   m_depth;
  void    KillParticles();
  GLuint  m_texture;
};

CRain::CRain(int numParticles, CVector origin, CObject *MyBucket,
             float height, float width, float depth)
  : m_height(height), m_width(width), m_depth(depth), CParticleSystem
(numParticles, origin, MyBucket)
{
}

void CRain::InitializeParticle(CParticle *particleToInitialize)
{
  particleToInitialize->m_pos.y = m_height;
  particleToInitialize->m_pos.x = m_origin.x + FRAND * m_width;
  particleToInitialize->m_pos.z = m_origin.z + FRAND * m_depth;

  particleToInitialize->m_size = DROP_SIZE;

  particleToInitialize->m_velocity.x = DROP_VELOCITY.x ;
particleToInitialize->m_velocity.y = DROP_VELOCITY.y + FRAND * 0.5;
  particleToInitialize->m_velocity.z = DROP_VELOCITY.z ;
  Attach(particleToInitialize);
}


void CRain::Update(float elapsedTime)
{
  CParticle *tempParticle;
  if(HasChild())
  {
    tempParticle = (CParticle *)childNode;
   for(int iLoop = 1; iLoop <= CountNodes() -1; iLoop++)
    {
```

```
      tempParticle->m_pos = tempParticle->m_pos +
                 (tempParticle->m_velocity * elapsedTime);
      tempParticle = (CParticle *)tempParticle->nextNode;
    }
  }

  m_accumulatedTime += elapsedTime;

  float newParticles = DROPS_PER_SEC * m_accumulatedTime;

  m_accumulatedTime -= 1.0f/(float)DROPS_PER_SEC * newParticles;

      KillParticles();
  Emit(newParticles);
}


void CRain::KillParticles(){
   if(HasChild())
   {
     CParticle *tempParticle, *tempParticle1;
     tempParticle = NULL;
     tempParticle = (CParticle *)childNode;
     while(tempParticle!=NULL)
     {
       if ((tempParticle->m_pos.y <= 0))
       {
         ((CBucket *)m_ParticleBucket)->GetParticle(tempParticle);
          tempParticle = (CParticle *)childNode;
          m_numParticles--;
       }
       else
       {
         tempParticle = (CParticle *)tempParticle->nextNode;
         if(tempParticle->IsLastChild())
         {
           tempParticle = (CParticle *)childNode;
           tempParticle = NULL;
         }
       }
     }
   }
}

void CRain::Render()
{
  CParticle *tempParticle;
  glEnable(GL_BLEND);
  glEnable(GL_TEXTURE_2D);

  glBlendFunc(GL_SRC_ALPHA, GL_ONE);

  glBindTexture(GL_TEXTURE_2D, m_texture);
  glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);

  CVector partPos;
  float size;
  glPushMatrix();
  //glTranslatef(m_origin.x, m_origin.y, m_origin.z);
  glBegin(GL_QUADS);
  if(HasChild())
```

```
  {
    tempParticle = (CParticle *)childNode;
    for(int iLoop = 1; iLoop <= CountNodes() -1; iLoop++)
    {
        glColor4f(0.0,0.0,1.0,0.6);
      partPos = tempParticle->m_pos;
      size = tempParticle->m_size / 2;
      glTexCoord2f(0.0, 1.0);
      glVertex3f(partPos.x - size, partPos.y + size, partPos.z);
      glTexCoord2f(1.0, 1.0);
      glVertex3f(partPos.x + size, partPos.y + size, partPos.z);
      glTexCoord2f(1.0, 0.0);
      glVertex3f(partPos.x + size, partPos.y - size, partPos.z);
      glTexCoord2f(0.0, 0.0);
      glVertex3f(partPos.x - size, partPos.y - size, partPos.z);
      tempParticle = (CParticle *)tempParticle->nextNode;
    }
   }
  glEnd();

  glPopMatrix();
  glDisable(GL_BLEND);
  glDisable(GL_TEXTURE_2D);
}

static char *drops[] = {
  "0000011111000000",
  "0000111111100000",
  "0000111111100000",
  "0000110111110000",
  "0000110111110000",
  "0000110111110000",
  "0000110111100000",
  "0000111111100000",
  "0000011111100000",
  "0000011111100000",
  "0000011111000000",
  "0000011111000000",
  "0000001111000000",
  "0000001111000000",
  "0000000110000000",
  "0000000110000000",
  "0000000110000000",
};

void CRain::InitializeSystem()
{
  glGenTextures(1, &m_texture);
  glBindTexture(GL_TEXTURE_2D, m_texture);
 GLubyte floorTexture[16][16][4];
  GLubyte *loc;
  int s, t;

  /* Setup RGB image for the texture. */
  loc = (GLubyte*) floorTexture;
  for (t = 0; t < 16; t++) {
    for (s = 0; s < 16; s++) {
      if (drops[t][s] == '1') {
        /* Nice blue. */
        loc[0] = 0x00;
        loc[1] = 0x00;
```

```
        loc[2] = 0x66;
        loc[3] = 0xff;
      } else {
        /* Black. */
        loc[0] = 0;
        loc[1] = 0;
        loc[2] = 0;
        loc[3] = 0;
      }
      loc += 4;
    }
  }

  glPixelStorei(GL_UNPACK_ALIGNMENT, 1);

  /*BITMAPINFOHEADER bitmapInfoHeader;
  unsigned char *buffer = LoadBitmapFileWithAlpha("rain.bmp",
&bitmapInfoHeader);       */

  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                  GL_LINEAR_MIPMAP_NEAREST);
  glTexImage2D(GL_TEXTURE_2D, 0, 4, 16, 16, 0, GL_RGBA,
                GL_UNSIGNED_BYTE, floorTexture);
  gluBuild2DMipmaps(GL_TEXTURE_2D, 4, 16, 16, GL_RGBA,
                GL_UNSIGNED_BYTE, floorTexture);

  CParticleSystem::InitializeSystem();
}

void CRain::KillSystem()
{
  if (glIsTexture(m_texture))
  {
    glDeleteTextures(1, &m_texture);
  }

  CParticleSystem::KillSystem();
}

#endif
```

1.8 smoke.h

```
#ifndef __SMOKE_H_
#define __SMOKE_H_

#include "particles.h"
#include <stdlib.h>


class CSmoke : public CParticleSystem
{
public:
  CSmoke(int maxParticles, CVector origin, CObject *MyBucket, float
energy, float alpha) ;

  void  Update(float elapsedTime);
```

```
  void  Render();

  void  InitializeSystem();
  void  KillSystem();

protected:
  float m_maxEnergy;
  float m_minAlpha;
  void  InitializeParticle(CParticle *particleToInitialize);
  void  KillParticles();
  GLuint  m_texture;
};

CSmoke::CSmoke(int numParticles, CVector origin, CObject *MyBucket,
float energy,float alpha)
: m_maxEnergy(energy), m_minAlpha(alpha), CParticleSystem(numParticles,
origin, MyBucket)
{

}

void CSmoke::InitializeParticle(CParticle *particleToInitialize)
{
    particleToInitialize->m_pos = m_origin;
    particleToInitialize->m_size = 0.1;

    particleToInitialize->m_velocity.x = 0.02 * (rand() % 3 - 1);
    particleToInitialize->m_velocity.y = 0.2;
    particleToInitialize->m_velocity.z = 0.02 * (rand() % 3 - 1);
    particleToInitialize->m_energy = (float)10.0 +
                           5.0 * ((rand() % 3) - 1);
    particleToInitialize->m_colorDelta[3] = 0.02 + m_maxEnergy * FRAND;

    Attach(particleToInitialize);
}




void CSmoke::Update(float elapsedTime)
{
  CParticle *tempParticle;
  if(HasChild())
  {
    tempParticle = (CParticle *)childNode;
   for(int iLoop = 1; iLoop <= CountNodes() -1; iLoop++)
    {
      tempParticle->m_pos = tempParticle->m_pos +
        (tempParticle->m_velocity * elapsedTime);
      tempParticle->m_energy -= 0.03f;
      tempParticle->m_colorDelta[3] += 0.01f;
      tempParticle->m_size += 0.003;
      tempParticle = (CParticle *)tempParticle->nextNode;
    }
  }
  m_accumulatedTime += elapsedTime;

  float newParticles = 5 * m_accumulatedTime;

 if(m_accumulatedTime >= 0.3)
 {
```

```
  m_accumulatedTime = 0.0;
 }
  KillParticles();
  Emit(newParticles);
}

void CSmoke::KillParticles()
{
   if(HasChild())
   {
     CParticle *tempParticle, *tempParticle1;
     tempParticle = NULL;
     tempParticle = (CParticle *)childNode;
     while(tempParticle!=NULL)
     {
       if ((tempParticle->m_energy <= 0))// ||
           (tempParticle->m_colorDelta[3] >= 100.0))
       {
         ((CBucket *)m_ParticleBucket)->GetParticle(tempParticle);
          tempParticle = (CParticle *)childNode;
          m_numParticles--;
       }
       else
       {
         tempParticle = (CParticle *)tempParticle->nextNode;
         if(tempParticle->IsLastChild())
         {
           tempParticle = NULL;
         }
       }
     }
   }
}


void CSmoke::Render()
{
  CParticle *tempParticle;
  glEnable(GL_BLEND);
  glEnable(GL_TEXTURE_2D);

  glBlendFunc(GL_SRC_ALPHA, GL_ONE);

  glBindTexture(GL_TEXTURE_2D, m_texture);
  glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);

  CVector partPos;
  float size;
  glPushMatrix();
  glTranslatef(m_origin.x, m_origin.y, m_origin.z);
  glBegin(GL_QUADS);
  if(HasChild())
  {
    tempParticle = (CParticle *)childNode;
    for(int iLoop = 1; iLoop <= CountNodes() -1; iLoop++)
    {
      partPos = tempParticle->m_pos;
      glColor4f(0.5,0.5,0.5,tempParticle->m_colorDelta[3]);
      size = tempParticle->m_size / 2;
      glTexCoord2f(0.0, 1.0);
      glVertex3f(partPos.x - size, partPos.y + size, partPos.z);
```

```
      glTexCoord2f(1.0, 1.0);
      glVertex3f(partPos.x + size, partPos.y + size, partPos.z);
      glTexCoord2f(1.0, 0.0);
      glVertex3f(partPos.x + size, partPos.y - size, partPos.z);
      glTexCoord2f(0.0, 0.0);
      glVertex3f(partPos.x - size, partPos.y - size, partPos.z);
      tempParticle = (CParticle *)tempParticle->nextNode;
    }
   }
  glEnd();
  glPopMatrix();
  glDisable(GL_BLEND);
  glDisable(GL_TEXTURE_2D);
}

void CSmoke::InitializeSystem()
{
  glGenTextures(1, &m_texture);
  glBindTexture(GL_TEXTURE_2D, m_texture);
  GLubyte smokeTexture[16][16][4];
  GLubyte *loc;
  int s, t;
```

```
 static char *smoke[] = {
  "1111100000011111",
  "1111000010001111",
  "1100000000100011",
  "1000010000000001",
  "0010000000010000",
  "0000101110100000",
  "0100011111000000",
  "0000101010100000",
  "0100111111100000",
  "0000001111000000",
  "0000001110000000",
  "0001000100001000",
  "1000000000000001",
  "1100000010000011",
  "1111001000001111",
  "1001110000011101",
};

  /* Setup RGB image for the texture. */
  loc = (GLubyte*) smokeTexture;
  for (t = 0; t < 16; t++) {
    for (s = 0; s < 16; s++) {
      if (smoke[t][s] == '0') {
        /* Nice blue. */
        loc[0] = 0x66;
        loc[1] = 0x66;
```

```
        loc[2] = 0x66;
        loc[3] = 0x66;
      } else {
        /* Light gray. */
        loc[0] = 0;
        loc[1] = 0;
        loc[2] = 0;
        loc[3] = 0;
      }
      loc += 4;
    }
  }

  glPixelStorei(GL_UNPACK_ALIGNMENT, 1);

  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                  GL_LINEAR_MIPMAP_NEAREST);
  glTexImage2D(GL_TEXTURE_2D, 0, 4, 16, 16, 0, GL_RGBA,
               GL_UNSIGNED_BYTE, smokeTexture);
  gluBuild2DMipmaps(GL_TEXTURE_2D, 4, 16, 16, GL_RGBA,
                    GL_UNSIGNED_BYTE, smokeTexture);

  CParticleSystem::InitializeSystem();

}

void CSmoke::KillSystem()
{
  if (glIsTexture(m_texture))
  {
    glDeleteTextures(1, &m_texture);
  }

  CParticleSystem::KillSystem();
}

#endif
```

1.9 snow.h

```
#ifndef __SNOW_H_
#define __SNOW_H_

#include "particles.h"

const CVector SNOWFLAKE_VELOCITY  (0.0f, -0.1f, 0.0f);
const CVector VELOCITY_VARIATION  (0.2f, 0.5f, 0.2f);
const float     SNOWFLAKE_SIZE      = 0.02f;
const float     SNOWFLAKES_PER_SEC  = 200;

class CSnowstorm : public CParticleSystem
{
public:
  CSnowstorm(int maxParticles, CVector origin, CObject *MyBucket, float
height, float width, float depth);

  void  Update(float elapsedTime);
  void  Render();
  void  InitializeSystem();
  void  KillSystem();

protected:
  void    InitializeParticle(CParticle *particleToInitialize);
  float   m_height;
  float   m_width;
  float   m_depth;
  void    KillParticles();
  GLuint  m_texture;
};

CSnowstorm::CSnowstorm(int maxParticles, CVector origin, CObject
*MyBucket, float height, float width, float depth)
  : m_height(height), m_width(width), m_depth(depth), CParticleSystem
(maxParticles, origin, MyBucket)
{
}

void CSnowstorm::InitializeParticle(CParticle *particleToInitialize)
{
  particleToInitialize->m_pos.y = m_height;
  particleToInitialize->m_pos.x = m_origin.x + FRAND * m_width;
  particleToInitialize->m_pos.z = m_origin.z + FRAND * m_depth;

  particleToInitialize->m_size = SNOWFLAKE_SIZE;

  particleToInitialize->m_velocity.x = SNOWFLAKE_VELOCITY.x +
                       FRAND * VELOCITY_VARIATION.x;
  particleToInitialize->m_velocity.y = SNOWFLAKE_VELOCITY.y +
                       FRAND * VELOCITY_VARIATION.y;
  particleToInitialize->m_velocity.z = SNOWFLAKE_VELOCITY.z +
                       FRAND * VELOCITY_VARIATION.z;

  Attach(particleToInitialize);
}

void CSnowstorm::Update(float elapsedTime)
{
  CParticle *tempParticle;
```

```
  if(HasChild())
  {
    tempParticle = (CParticle *)childNode;
   for(int iLoop = 1; iLoop <= CountNodes() -1; iLoop++)
    {
      tempParticle->m_pos = tempParticle->m_pos +
                  (tempParticle->m_velocity * elapsedTime);
      tempParticle = (CParticle *)tempParticle->nextNode;
    }
  }


  m_accumulatedTime += elapsedTime;

  float newParticles = SNOWFLAKES_PER_SEC * m_accumulatedTime;

  m_accumulatedTime -= 1.0f/(float)SNOWFLAKES_PER_SEC * newParticles;

  KillParticles();
  Emit(newParticles);
}

void CSnowstorm::KillParticles(){
   if(HasChild())
   {
     CParticle *tempParticle, *tempParticle1;
     tempParticle = NULL;
     tempParticle = (CParticle *)childNode;
     while(tempParticle!=NULL)
     {
       if ((tempParticle->m_pos.y <= m_origin.y))
       {
         ((CBucket *)m_ParticleBucket)->GetParticle(tempParticle);
          tempParticle = (CParticle *)childNode;
          m_numParticles--;
       }
       else
       {
         tempParticle = (CParticle *)tempParticle->nextNode;
         if(tempParticle->IsLastChild())
         {
           tempParticle = (CParticle *)childNode;
           tempParticle = NULL;
         }
       }
     }
   }
}




void CSnowstorm::Render()
{
  CParticle *tempParticle;

  glEnable(GL_BLEND);
  glEnable(GL_TEXTURE_2D);

  glBlendFunc(GL_SRC_ALPHA, GL_ONE);
```

```
  glBindTexture(GL_TEXTURE_2D, m_texture);
  glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);

  CVector partPos;
  float size;
  glPushMatrix();
  //glTranslatef(m_origin.x, m_origin.y, m_origin.z);
  glBegin(GL_QUADS);
   if(HasChild())
  {
    tempParticle = (CParticle *)childNode;
    for(int iLoop = 1; iLoop <= CountNodes() -1; iLoop++)
    {
        glColor3f(1.0,1.0,1.0);
      partPos = tempParticle->m_pos;
      size = tempParticle->m_size / 2;
      glTexCoord2f(0.0, 1.0);
      glVertex3f(partPos.x - size, partPos.y + size, partPos.z);
      glTexCoord2f(1.0, 1.0);
      glVertex3f(partPos.x + size, partPos.y + size, partPos.z);
      glTexCoord2f(1.0, 0.0);
      glVertex3f(partPos.x + size, partPos.y - size, partPos.z);
      glTexCoord2f(0.0, 0.0);
      glVertex3f(partPos.x - size, partPos.y - size, partPos.z);
      tempParticle = (CParticle *)tempParticle->nextNode;
    }
   }
  glEnd();

  glPopMatrix();
  glDisable(GL_BLEND);
  glDisable(GL_TEXTURE_2D);
}


void CSnowstorm::InitializeSystem()
{
  glGenTextures(1, &m_texture);
  glBindTexture(GL_TEXTURE_2D, m_texture);
  GLubyte snowTexture[16][16][4];
  GLubyte *loc;
  int s, t;









 static char *snow[] = {
  "1111101010111111",
  "1111110001111111",
  "1010111011101011",
  "1100111011100111",
  "1000111011100011",
  "1111010001011111",
  "0111101010111101",
```

```
  "1011010001011011",
  "0000000000000001",
  "1011010001011011",
  "0111101010111101",
  "1100010001000111",
  "1110011011001111",
  "1101011011010111",
  "1111110001111111",
  "1111101010111111",
};

  /* Setup RGB image for the texture. */
  loc = (GLubyte*) snowTexture;
  for (t = 0; t < 16; t++) {
    for (s = 0; s < 16; s++) {
      if (snow[t][s] == '0') {
        /* Nice blue. */
        loc[0] = 0x66;
        loc[1] = 0x66;
        loc[2] = 0x66;
        loc[3] = 0x66;
      } else {
        /* Light gray. */
        loc[0] = 0;
        loc[1] = 0;
        loc[2] = 0;
        loc[3] = 0;
      }
      loc += 4;
    }
  }

  glPixelStorei(GL_UNPACK_ALIGNMENT, 1);

  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
GL_LINEAR_MIPMAP_NEAREST);
  glTexImage2D(GL_TEXTURE_2D, 0, 4, 16, 16, 0, GL_RGBA,
GL_UNSIGNED_BYTE, snowTexture);
  gluBuild2DMipmaps(GL_TEXTURE_2D, 4, 16, 16, GL_RGBA,
GL_UNSIGNED_BYTE, snowTexture);

  CParticleSystem::InitializeSystem();
}

void CSnowstorm::KillSystem()
{
  if (glIsTexture(m_texture))
  {
    glDeleteTextures(1, &m_texture);
  }

  CParticleSystem::KillSystem();
}

#endif
```

1.10 general.hpp

```
#ifndef _GENERAL_HPP_
#define _GENERAL_HPP_
#include "vectorlib.hpp"

class CNode
{
      public:
            CNode *parentNode;
            CNode *childNode;
            CNode *prevNode;
            CNode *nextNode;

            bool HasParent();
            bool HasChild();
            bool IsFirstChild();
            bool IsLastChild();
            void AttachTo(CNode *NewParent);
            void Attach(CNode *newChild);
            void Detach();
            int CountNodes();
            CNode();
            CNode(CNode *Node);
            virtual ~CNode();

};


#endif
```

1.11 vectorlib.hpp

```
#ifndef __VECTOR_H
#define __VECTOR_H

#include <math.h>
#include <stdlib.h>

/*
     VECTOR.H

     CVector class

     OpenGL Game Programming
     by Kevin Hawkins and Dave Astle

     Some operators of the CVector class based on
     operators of the CVector class by Bas Kuenen.
     Copyright (c) 2000 Bas Kuenen. All Rights Reserved.
     homepage: baskuenen.cfxweb.net
*/

#define PI         (3.14159265359f)
#define DEG2RAD(a)       (PI/180*(a))
#define RAD2DEG(a)       (180/PI*(a))
#define FRAND   (((float)rand()-(float)rand())/RAND_MAX)
#define Clamp(x, min, max)  x = (x<min  ? min : x<max ? x : max);

#define SQUARE(x)  (x)*(x)


typedef float scalar_t;

class CVector
{
public:
     scalar_t x;
     scalar_t y;
     scalar_t z;    // x,y,z coordinates

public:
     CVector(scalar_t a = 0, scalar_t b = 0, scalar_t c = 0) : x(a), y
(b), z(c) {}
     CVector(const CVector &vec) : x(vec.x), y(vec.y), z(vec.z) {}

      // vector index
      scalar_t &operator[](const long idx)
      {
            return *((&x)+idx);
      }

     // vector assignment
     const CVector &operator=(const CVector &vec)
     {
          x = vec.x;
          y = vec.y;
          z = vec.z;

          return *this;
     }
```

```
// vecector equality
const bool operator==(const CVector &vec) const
{
     return ((x == vec.x) && (y == vec.y) && (z == vec.z));
}

// vecector inequality
const bool operator!=(const CVector &vec) const
{
     return !(*this == vec);
}

// vector add
const CVector operator+(const CVector &vec) const
{
     return CVector(x + vec.x, y + vec.y, z + vec.z);
}

// vector add (opposite of negation)
const CVector operator+() const
{
     return CVector(*this);
}

// vector increment
const CVector& operator+=(const CVector& vec)
{    x += vec.x;
     y += vec.y;
     z += vec.z;
     return *this;
}

// vector subtraction
const CVector operator-(const CVector& vec) const
{
     return CVector(x - vec.x, y - vec.y, z - vec.z);
}

// vector negation
const CVector operator-() const
{
     return CVector(-x, -y, -z);
}

// vector decrement
const CVector &operator-=(const CVector& vec)
{
     x -= vec.x;
     y -= vec.y;
     z -= vec.z;

     return *this;
}

// scalar self-multiply
const CVector &operator*=(const scalar_t &s)
{
     x *= s;
     y *= s;
     z *= s;
```

```
          return *this;
     }

     // scalar self-divecide
     const CVector &operator/=(const scalar_t &s)
     {
          const float recip = 1/s; // for speed, one divecision

          x *= recip;
          y *= recip;
          z *= recip;

          return *this;
     }

     // post multiply by scalar
     const CVector operator*(const scalar_t &s) const
     {
          return CVector(x*s, y*s, z*s);
     }

     // pre multiply by scalar
     friend inline const CVector operator*(const scalar_t &s, const
CVector &vec)
     {
          return vec*s;
     }

      const CVector operator*(const CVector& vec) const
      {
            return CVector(x*vec.x, y*vec.y, z*vec.z);
      }

      // post multiply by scalar
     /*friend inline const CVector operator*(const CVector &vec, const
scalar_t &s)
     {
          return CVector(vec.x*s, vec.y*s, vec.z*s);
     }*/

    // divide by scalar
     const CVector operator/(scalar_t s) const
     {
          s = 1/s;

          return CVector(s*x, s*y, s*z);
     }

     // cross product
     const CVector CrossProduct(const CVector &vec) const
     {
          return CVector(y*vec.z - z*vec.y, z*vec.x - x*vec.z, x*vec.y
- y*vec.x);
     }

     // cross product
     const CVector operator^(const CVector &vec) const
     {
          return CVector(y*vec.z - z*vec.y, z*vec.x - x*vec.z, x*vec.y
- y*vec.x);
```

```
}

// dot product
const scalar_t DotProduct(const CVector &vec) const
{
     return x*vec.x + y*vec.x + z*vec.z;
}

// dot product
const scalar_t operator%(const CVector &vec) const
{
     return x*vec.x + y*vec.x + z*vec.z;
}

// length of vector
const scalar_t Length() const
{
     return (scalar_t)sqrt((double)(x*x + y*y + z*z));
}

// return the unit vector
const CVector UnitVector() const
{
     return (*this) / Length();
}

// normalize this vector
void Normalize()
{
     (*this) /= Length();
}

const scalar_t operator!() const
{
     return sqrtf(x*x + y*y + z*z);
}

// return vector with specified length
const CVector operator | (const scalar_t length) const
{
     return *this * (length / !(*this));
}

// set length of vector equal to length
const CVector& operator |= (const float length)
{
     return *this = *this | length;
}

// return angle between two vectors
const float Angle(const CVector& normal) const
{
     return acos(*this % normal);
}

// reflect this vector off surface with normal vector
const CVector Reflection(const CVector& normal) const
{
     const CVector vec(*this | 1);     // normalize this vector
     return (vec - normal * 2.0 * (vec % normal)) * !*this;
}
```

```
      // rotate angle degrees about a normal
      const CVector Rotate(const float angle, const CVector& normal)
const
      {
            const float cosine = cosf(angle);
            const float sine = sinf(angle);

            return CVector(*this * cosine + ((normal * *this) * (1.0f -
cosine)) *
                           normal + (*this ^ normal) * sine);
      }
};

#endif
```

2. Archivos de implementación

2.1 bucket.cpp

```
#include "bucket.h"
#include <stdio.h>
CBucket::CBucket(int maxParticles)
{
 CParticle *tempParticle;
  while(maxParticles >=1)
  {
    tempParticle = new CParticle;
    if(tempParticle != NULL)
    {
       Attach((CNode *)tempParticle);
    }
     maxParticles--;
  }
}

CParticle * CBucket::GiveParticle()
{
       return((CParticle *)childNode);
}

void  CBucket::GetParticle(CParticle *Particle)
{
      Particle->Detach();
      Attach((CNode *)Particle);
}
```

2.2 general.cpp

```
#include "general.hpp"
bool CNode::HasParent()
{
	return(parentNode != NULL);
}

bool CNode::HasChild()
{
	return (childNode != NULL);
}

bool CNode::IsFirstChild()
{
	if(parentNode)
	{
		return(parentNode->childNode == this);
	}
	else
	{
		return(false);
	}
}

bool CNode::IsLastChild()
{
	if(parentNode)
	{
		return(parentNode->childNode->prevNode == this);
	}
	else
	{
		return(false);
	}
}

void CNode::AttachTo (CNode *newParent)
{
	if(parentNode)
	{
		Detach();
	}
	parentNode = newParent;
	if(parentNode->childNode)
	{
		prevNode = parentNode->childNode->prevNode;
		nextNode = parentNode->childNode;
		parentNode->childNode->prevNode->nextNode = this;
	}
	else
	{
		parentNode->childNode = this;
	}

}

void CNode::Attach(CNode *newChild)
{
	if(newChild->HasParent())
```

```
	{
		newChild->Detach();
	}
	newChild->parentNode = this;
	if(childNode)
	{
		newChild->prevNode = childNode->prevNode;
		newChild->nextNode = childNode;
		childNode->prevNode->nextNode = newChild;
		childNode->prevNode = newChild;
	}
	else
	{
		childNode = newChild;
	}
}

void CNode::Detach()
{
	if(parentNode && parentNode->childNode == this)
	{
		if(nextNode != this)
		{
			parentNode->childNode = nextNode;

			prevNode->nextNode = nextNode;
			nextNode->prevNode = prevNode;

			prevNode = this;
			nextNode = this;
		}
		else
		{
			parentNode->childNode = NULL;
		}
	  }
    else
    {
			prevNode->nextNode = nextNode;
			nextNode->prevNode = prevNode;

			prevNode = this;
			nextNode = this;
		parentNode = NULL;
    }
}

int CNode::CountNodes()
{
    int iCountChild = 0, iCountSiblings = 0;
      if (HasChild())
		iCountChild = childNode->CountNodes();

      if (HasParent() && !IsLastChild())
		iCountSiblings = nextNode->CountNodes();

    return(iCountChild + iCountSiblings + 1);
}

CNode::CNode()
{
```

```
	parentNode = childNode = NULL;
	prevNode = nextNode = this;
}

CNode::CNode(CNode *Node)
{
	parentNode = childNode = NULL;
	prevNode = nextNode = this;
	AttachTo(Node);
}

CNode::~CNode()
{
	Detach();
	while(childNode)
	{
		delete childNode;
	}
}
```

2.3 main.cpp

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <gl/glut.h>
#include "bucket.h"
#include "smoke.h"
#include "rain.h"
#include "snow.h"
#include "explotion.h"
#include "fireworks.h"
#include "psm.h"


void init(void);
void display(void);
void changeSize(int w, int h);
void processNormalKeys(unsigned char key, int x, int y);
void processSpecialKeys(int key, int x, int y);

int secs=0;
float x=0.0f,y=1.75f,z=-5.0f;
float lx=0.0f,ly=0.0f,lz=4.0f;
CPsm PSM;
CBucket *MyBucket;
CSmoke *g_smoke;
CRain *g_rain;
CSnowstorm *g_snow;
CExplotion *g_explotion;
CFireworks *g_fireworks;

int main(int argc, char** argv)
{
      //Inicializacion de GLUT
      glutInit (&argc, argv) ;
      glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB) ;
      glutInitWindowSize (500, 500) ;
      glutInitWindowPosition (100, 100) ;
      glutCreateWindow ("Particle Systems Manager") ;
      //Inicializaacion del Systema

      init() ;

      //Funciones
      // Funcion de render
      glutDisplayFunc (display);
      // Funcion Idle
      glutIdleFunc(display);
      //Funcion de Redimencion
      glutReshapeFunc(changeSize);
      //Teclas Normales
      glutKeyboardFunc(processNormalKeys);
      //Teclas dede Funcion
      glutSpecialFunc(processSpecialKeys);

      //Lopop Principal
      glutMainLoop () ;
      return 0 ;
}
```

```
void init(void)
{
      glClearColor(0.5f, 0.5f, 0.5f, 0.0f) ;
      glShadeModel(GL_SMOOTH) ;

      glClearColor(0.1, 0.1, 0.1, 1.0);
      gluLookAt(x, y, z,
                  x + lx,y + ly,z + lz,
                    0.0f,1.0f,0.0f);

    MyBucket = new CBucket(1500);
    g_smoke = new CSmoke(500, CVector(1.0, 0.0, 5.0),
             (CObject *)MyBucket, 5.0, 0.5);
    g_smoke->InitializeSystem();
    g_rain = new CRain(1000, CVector(0.0, 0.0, -3.0),
             (CObject *)MyBucket, 3.0, 4.0, 2.0);
    g_rain->InitializeSystem();
    g_snow = new CSnowstorm(500, CVector(0.0, 0.0, -4.0),
             (CObject *)MyBucket, 3.0, 4.0, 2.0);
    g_snow->InitializeSystem();
    g_explotion = new CExplotion(500, CVector(1.0, 0.0, 5.0),
            (CObject *)MyBucket, 15.0, 0.5);
    g_explotion->InitializeSystem();
    g_fireworks = new CFireworks(500, CVector(0.0, 3.0, 3.0),
            (CObject *)MyBucket);
    g_fireworks->InitializeSystem();
}

void display(void)
{     system("cls");
      printf("Bucket %d\n",MyBucket->CountNodes());
      static float tpassed = 0;
      glClear(GL_COLOR_BUFFER_BIT) ;
      glLoadIdentity();
      gluLookAt(x, y, z,
                  x + lx,y + ly,z + lz,
                    0.0f,1.0f,0.0f);
      secs = glutGet(GLUT_ELAPSED_TIME);


      // Draw ground
      glColor3f(0.3f, 0.3f, 0.3f);
      glBegin(GL_QUADS);
            glVertex3f(-100.0f, 0.0f, -100.0f);
            glVertex3f(-100.0f, 0.0f,  100.0f);
            glVertex3f( 100.0f, 0.0f,  100.0f);
            glVertex3f( 100.0f, 0.0f, -100.0f);
      glEnd();

    glPushMatrix();
                  g_smoke->Render();
                  g_smoke->Update((secs - tpassed)/1000);
                  g_rain->Render();
                  g_rain->Update((secs - tpassed)/1000);
                  g_snow->Render();
                  g_snow->Update((secs - tpassed)/1000);
                  g_explotion->Render();
                  g_explotion->Update((secs - tpassed)/1000);
                  g_fireworks->Render();
```

```
                    g_fireworks->Update((secs - tpassed)/1000);
                    g_explotion->MoveEmmit(CVector(-1.0, 0.0, 3.0));
                    g_smoke->MoveEmmit(CVector(-1.0, 0.0, 3.0));
                    g_explotion->MoveEmmit(CVector(1.0, 0.0, 3.0));
                    g_smoke->MoveEmmit(CVector(1.0, 0.0, 3.0));
                    g_explotion->MoveEmmit(CVector(-1.0, 0.0, -3.0));
                    g_smoke->MoveEmmit(CVector(-1.0, 0.0, -3.0));
                    g_explotion->MoveEmmit(CVector(1.0, 0.0, -3.0));
                    g_smoke->MoveEmmit(CVector(1.0, 0.0, -3.0));
    glPopMatrix();
      glutSwapBuffers();
  tpassed = secs;
}

void changeSize(int w, int h) {

      // Prevent a divide by zero, when window is too short
      // (you cant make a window of zero width).
      if(h <= 0)
            h = 1;

      float ratio = 1.0* w / h;

      // Reset the coordinate system before modifying
      glMatrixMode(GL_PROJECTION);
      glLoadIdentity();

      // Set the viewport to be the entire window
      glViewport(0, 0, w, h);

      // Set the correct perspective.
      gluPerspective(45,ratio,1,1000);
      glMatrixMode(GL_MODELVIEW);
      glLoadIdentity();
      gluLookAt(0.0,10.0,5.0,
                  0.0,10.0,-1.0,
                    0.0f,1.0f,0.0f);
}

void processNormalKeys(unsigned char key, int x, int y) {

      if (key == 27)
            exit(0);
}




void processSpecialKeys(int key, int x, int y) {

      switch(key) {
            case GLUT_KEY_F1 :
            g_smoke->InitializeSystem();
                  break;
            case GLUT_KEY_F2 :
                  break;
            case GLUT_KEY_F3 :
                  break;
      }
```

```
}
```

2.4 Objeto.cpp

```
#include "Objeto.h"
void CObject::Draw(){
	glPushMatrix();
		OnDraw();
		if (HasChild())
			((CObject*)childNode)->Draw();
	glPopMatrix();

	if (HasParent() && !IsLastChild())
		((CObject*)nextNode)->Draw();
}

void CObject::Animate(scalar_t deltaTime)
{
	OnAnimate(deltaTime);

	if (HasChild())
		((CObject*)childNode)->Animate(deltaTime);

	if (HasParent() && !IsLastChild())
		((CObject*)nextNode)->Animate(deltaTime);

	if (bDelete)
		delete this;
}

void CObject::Prepare()
{
	OnPrepare();

	if (HasChild())
		((CObject*)childNode)->Prepare();

	if (HasParent() && !IsLastChild())
		((CObject*)nextNode)->Prepare();
}

void CObject::OnAnimate(scalar_t deltaTime)
{
	position += velocity * deltaTime;
	velocity += acceleration * deltaTime;
}
```

2.5 particles.cpp

```
#include "particles.h"
#include "bucket.h"
#include <stdio.h>

CParticleSystem::CParticleSystem(int maxParticles, CVector origin,
CObject *MyBucket)
{
  m_maxParticles = maxParticles;
  m_origin = origin;
  m_ParticleBucket = MyBucket;
}

int CParticleSystem::Emit(int numParticles)
{
  CParticle *particleToInitialize;
  while (numParticles && (m_numParticles < m_maxParticles))
  {
    particleToInitialize = ((CBucket *)m_ParticleBucket)->GiveParticle
();
    if(particleToInitialize != NULL)
    {
      InitializeParticle(particleToInitialize);
      m_numParticles++;
    }
    --numParticles;
  }
  return numParticles;
}

void CParticleSystem::InitializeSystem()
{
 while(HasChild())
  {
    ((CBucket *)m_ParticleBucket)->GetParticle((CParticle *)
childNode);//childNode->AttachTo((CNode *)m_ParticleBucket);
  }
  m_numParticles = 0;
  m_accumulatedTime = 0.0f;
}

void CParticleSystem::KillSystem()
{

  m_numParticles = 0;
}

void CParticleSystem::MoveEmmit(CVector origin)
{
  CVector TempVector;
  TempVector = m_origin;
  m_origin =  origin;
  Render();
  m_origin = TempVector;
}

void CParticleSystem::MoveOrigin(CVector origin)
{
  m_origin =  origin;
```

```
}
```

2.6 psm.cpp

```
#include "psm.h"
#include <stdio.h>
void CPsm::Render()
{
  CParticleSystem *tempParticleSystem;
  tempParticleSystem = NULL;
  if (HasChild())
    {
      tempParticleSystem = (CParticleSystem *)childNode;
      do
      {
        tempParticleSystem->Render();
        tempParticleSystem = (CParticleSystem *)tempParticleSystem-
>nextNode;
      }while(!tempParticleSystem->IsLastChild());
    }
}

void CPsm::Update(float elapsedTime)
{
  CParticleSystem *tempParticleSystem;
  tempParticleSystem = NULL;
  if (HasChild())
    {
      tempParticleSystem = (CParticleSystem *)childNode;
      do
      {
        tempParticleSystem->Update(elapsedTime);
        tempParticleSystem = (CParticleSystem *)tempParticleSystem-
>nextNode;
      }while(!tempParticleSystem->IsLastChild());
    }
}
```

ANEXO IV

Instalación de glut para BloodShed Dev-C++ Win32

La mayoría de versiones de OS Linux traen instaladas las librerías de glut, no así las versiones de OS Microsoft. Para instalar GLUT32 en una plataforma Win32 (Microsoft), es necesario:

Para instalar:

1) abrir el archivo **glut32DevC.zip**

2) copiar el archivo **glut.h** al subdirectorio **Include/GL** del directorio del compilador.

3) copiar **glut32.dll** al fólder **System32** de su sistema operativo, o copiarlo al fólder del proyecto.

4) Copiar **libglut32.a** al fólder **lib** del directorio del compilador.

Para utilizarlo, se debe:

1) Crear un nuevo proyecto.

2) ir a **Project -> Project Options**.

3) bajo **Linker Options/Optional Libs** o **Object files** escribir **"-lopengl32 –lglut32 –glu32**"

CONCLUSIONES

- El diseño modular de las aplicaciones facilita su desarrollo individual e independiente, ya que al tomar cada módulo como una caja negra que recibe ciertos datos y devuelve otros, permite la abstracción máxima de cada objeto, y vuelve más eficiente la implementación.

- La implementación de sistemas de partículas se vuelve sumamente sencilla, siempre y cuando se posea la teoría y los ejemplos necesarios para ello, y en este trabajo se ha presentado una forma lógica de generación y manipulación de los mismos.

- Al desarrollar en un entorno que no posee recolección de basura, como es C/C++, debe existir un método eficiente de reutilización de la misma, para evitar desbordamientos y pérdida de objetos durante la ejecución de las aplicaciones; en este caso, el balde de partículas permite controlar el número máximo de partículas utilizadas por el conjunto de sistemas y la reutilización de las mismas, permitiendo un control del flujo de memoria en el momento de la animación.

- Para que el código sea multiplataforma, es necesario apegarse lo más que se pueda a las normas ISO C 99 e ISO C++ 98, pero se debe tener en cuenta que no todos los compiladores cumplen con esos estandares, por lo que, además del diseño del código, deben analizarse los compiladores a utilizar en cada plataforma, con el objeto de evitar cambios drásticos al código al momento de hacer el puerto a otra plataforma.

- Con la aproximación utilizada para la implementación de las partículas, sus sistemas y sus manejadores, se puede ampliar y facilitar la implementación de una máquina de modelado de mundos virtuales.

BIBLIOGRAFIA

- William T. Reeves, Particle Systems - *A Technique for Modeling a Class of Fuzzy Objects"*, Computer Graphics 17:3 pp. 359-376, 1983 (SIGGRAPH 83).

- Dave Astle, Kevin Hawkins, Andre LaMothe: *OpenGL Game Programming*; Prima Publishing; Book and CD edition (2000)

- Jeff Lander: The Ocean Spray in Your Face, Game Developer Magazine, July 1998.

- John van der Burg: Building an Advanced Particle System, Game Developer Magazine, March 2000.

INFORMACION ADICIONAL

Sitios Web

- GamaSutra
  http://www.gamasutra.com

- GameDev.net:
  http://www.gamedev.net/

- Java Games Programming Index Page:
  http://fivedots.coe.psu.ac.th/~ad/jg/index.html

- NeHe Productions:
  http://nehe.gamedev.net/

- Particle Systems API:
  http://www.cs.unc.edu/~davemc/Particle/

- Particle Systems:
  http://www.cs.wpi.edu/~matt/courses/cs563/talks/psys.html

- Particle Systems – ResearchIndex document query:
  http://citeseer.nj.nec.com/cs?q=Particle+Systems&cs=1

- The Game Programming and Design Search Engine
  http://www.gdse.com

Libros

- Andre LaMoth: *Tricks of the Windows Game Programming Gurus*; SAMS (October 1999)

- David H. Eberly: *3D Game Engine Design: A Practical Approach to Real-Time Computer Graphics*; Morgan Kaufmann; Book and CD-ROM edition (September 2000)

- Jesse Liberty, Vishwajit Aklecha : *C++ Unleashed*; Sams; Book and CD-ROM edition (November 1998)

- Kurt Wall, Mark Watson, Mark Whitis: *Linux Programming Unleashed*; SAMS (August 1999)

- Tom McReynolds, David Blythe: *Advanced Graphics Programming Techniques Using OpenGL*; SIGGRAPH '98 Course, Silicon Graphics (April 1998)

- Tomas Moller, Eric Haines: *Real-Time Rendering*; A K Peters Ltd; 1 edition (June 1999)