Universidad de Huelva

Tema 11

Partículas y técnicas de animación

Índice

Índice

- Hasta el momento todos las escenas que hemos generado están compuestas de figuras rígidas. Las superficies de estas figuras están definidas en base a vértices que no cambian de posición.
- Siguiendo las técnicas vistas hasta ahora, para considerar superficies no rígidas (por ejemplo una onda) habría que modificar los atributos de los vértices. Es decir, habría que acceder a la memoria de la GPU para modificar el contenido de los VBO que almacenan los atributos.
- Para desarrollar esto, antes de cada proceso de renderizado habría que acceder a la memoria de la GPU para modificar los datos. Esto es algo muy costoso en tiempo.

- Una opción alternativa consiste en introducir el tiempo como una variable uniforme y utilizarlo para deformar los atributos de los vértices dentro de los shaders.
- Otra posibilidad consiste en modificar los atributos en función de los valores generados en un renderizado anterior. Para eso es necesario almacenar los datos generados en los shaders en la memoria de la GPU y ser capaz de leer estos datos en rederizados posteriores.
- Esta última característica fue incluida en la versión OpenGL 4.0 y se conoce como ***Transform Feedback***.

- Vamos a mostrar un ejemplo de superficie flexible basado en la introducción del tiempo como variable uniforme.
- Se trata de generar una superficie ondulada en la que las ondas se desplacen lateralmente.
- Para ello se generará una superficie cuadrada por medio de numerosas primitivas y se calculará la altura de cada punto por medio de una función temporal.
- La ecuación de una onda es la siguiente:

$$y = A * \text{sen}\ (\ 2\pi/\lambda \cdot (x - v\ t)\ )$$

donde A es la amplitud, λ es la longitud de onda y v es la velocidad de desplazamiento de la onda. El factor $2\pi/\lambda$ se conoce también como número de onda.

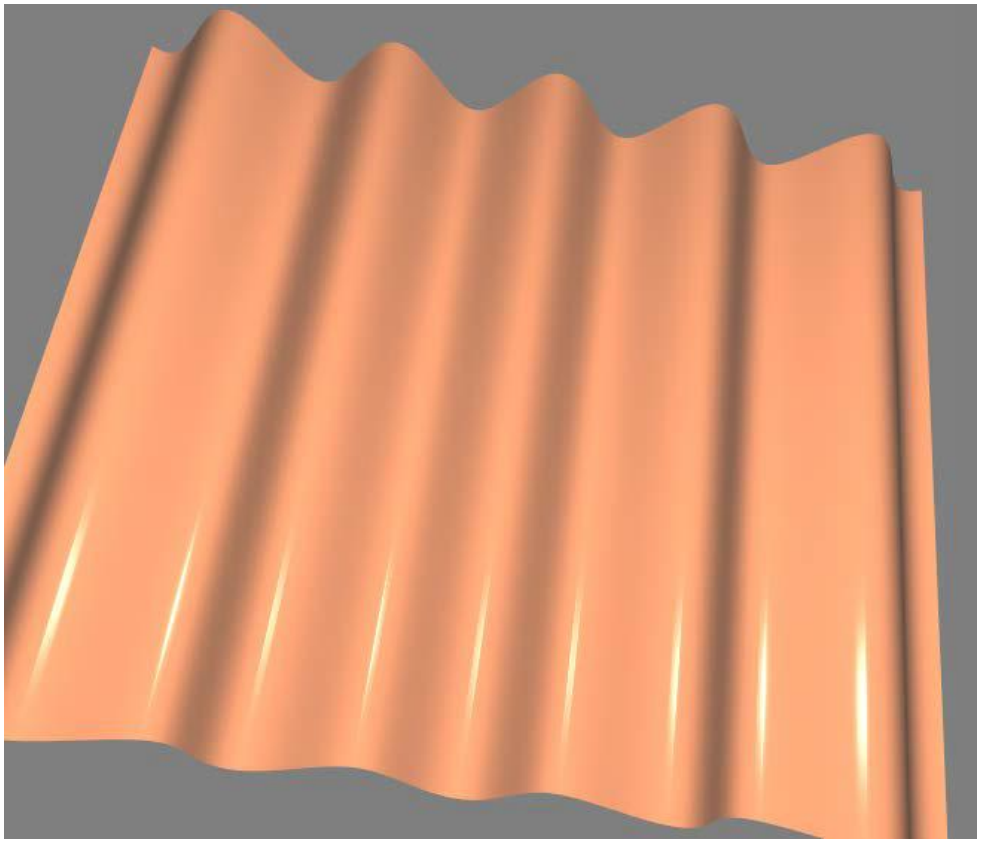
- VertexShader

```
#version 400

layout (location = 0) in vec3 VertexPosition;

out vec4 Position;
out vec3 Normal;

uniform float Time;     // The animation time
uniform float K;        // Wavenumber
uniform float Velocity; // Wave's velocity
uniform float Amp;      // Wave's amplitude

uniform mat4 ModelViewMatrix;
uniform mat3 NormalMatrix;
uniform mat4 MVP;

…
```

- VertexShader

```
…
void main()
{
  vec4 pos = vec4(VertexPosition,1.0);

  // Translate the y coordinate
  float u = K * (pos.x - Velocity * Time);
  pos.y = Amp * sin( u );

  // Compute the normal vector
  vec3 n = vec3(0.0);
  n.xy = normalize(vec2(-K*Amp*cos( u ), 1.0));

  Position = ModelViewMatrix * pos;
  Normal = NormalMatrix * n;
  gl_Position = MVP * pos;
}
```

- Ejemplo de imagen generada en un instante (la onda se va desplazando a la derecha).



Índice

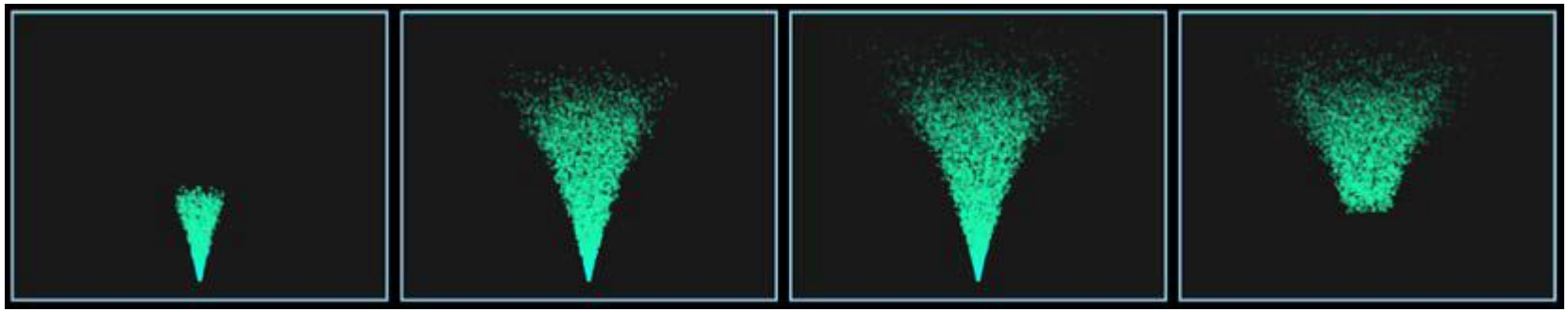
- Los sistemas de partículas permiten modelar fenómenos difusos como el fuego, humo, sprays o explosiones.
- Los modelos están formados por un gran número de partículas que se suelen representar como puntos (primitivas GL_POINTS), aunque también se pueden representar como cuadrados (*point sprites*) o incluso figuras rígidas si es necesario que cambien de tamaño con la distancia.
- Cada partícula tiene un ciclo de vida: nace, se mueve en función de ciertas leyes y finalente muere.
- Para generar un proceso continuo es necesario "resucitar" o "reciclar" las partículas, es decir, producir una nueva partícula cuando una partícula termina su ciclo de vida.

- La forma más simple de generar un sistema de partículas es considerar un conjunto de vértices donde cada vértice representa una partícula y dibujar la figura como primitivas GL_POINTS.
- Para generar el movimiento se incluye el tiempo como variable uniforme.
- Todas las partículas se mueven siguiendo la misma regla. Por ejemplo, una fuente de partículas lanzadas hacia arriba y que sufren la aceleración de la gravedad seguiría la ecuación:

$$\mathbf{p}(t) = \mathbf{p}_0 + \mathbf{v}_0 \cdot t + \mathbf{a} \cdot t^2$$

donde $\mathbf{p}$ es el vector de posición, $\mathbf{v}_0$ es la velocidad inicial y $\mathbf{a}$ es la aceleración.

- El siguiente ejemplo muestra una fuente de partículas que parten de la misma posición (p0) y tienen una velocidad inicial generada aleatoriamente. Para que no nazcan todas al mismo tiempo se considera también un valor que marque el instante de nacimiento de la partícula.
- Cada partícula tiene, por tanto, dos atributos: velocidad inicial e instante de nacimiento. El código a ejecutar en la CPU debe generar estos valores aleatorios y cargarlos en los correspondientes VBO.



- Código

```
vec3 v(0.0f);
float velocity, theta, phi;

GLfloat *data = new GLfloat[nParticles * 3];
for(int i = 0; i < nParticles; i++ )
{
  theta = (Glfloat) (M_PI * randFloat() / 6.0);
  phi = (Glfloat) (2.0 *M_PI * randFloat());
  v.x = sinf(theta) * cosf(phi);
  v.y = cosf(theta);
  v.z = sinf(theta) * sinf(phi);

  velocity = 1.25f + 0.25f * randFloat();
  v = v * velocity;
  data[3*i] = v.x;
  data[3*i+1] = v.y;
  data[3*i+2] = v.z;
}
…
```

- Código

```
…

int buffSize = nParticles * sizeof(float);
glBindBuffer(GL_ARRAY_BUFFER, initVel);
glBufferSubData(GL_ARRAY_BUFFER, 0, 3*buffSize,data);

float * data = new GLfloat[nParticles];
float time = 0.0f, rate = 0.00075f;
for( unsigned int i = 0; i < nParticles; i++ )
 { data[i] = time; time += rate;}

glBindBuffer(GL_ARRAY_BUFFER,startTime);
glBufferSubData(GL_ARRAY_BUFFER, 0, buffSize, data);

glDisable(GL_DEPTH_TEST);
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
glEnable(GL_POINT_SPRITE);
glPointSize(10.0f);
```

- VertexShader

```
#version 400

layout (location = 0) in vec3 VertexInitVel;
layout (location = 1) in float StartTime;

out float Transp;

uniform float Time;
uniform vec3 Gravity = vec3(0.0,-0.05,0.0);
uniform float ParticleLifetime;
uniform mat4 MVP;

…
```

- VertexShader

```
…
void main()
{
  vec3 pos = vec3(0.0);
  Transp = 0.0;

  if( Time > StartTime )
  {
    float t = Time - StartTime;
    if( t < ParticleLifetime )
    {
      pos = VertexInitVel * t + Gravity * t * t;
      Transp = 1.0 - t / ParticleLifetime;
    }
  }
  gl_Position = MVP * vec4(pos, 1.0);
}
```

- FragmentShader

```
#version 400

in float Transp;

uniform sampler2D ParticleTex;

layout ( location = 0 ) out vec4 FragColor;

void main()
{
  FragColor = texture(ParticleTex,gl_PointCoord);
  FragColor.a *= Transp;
}
```

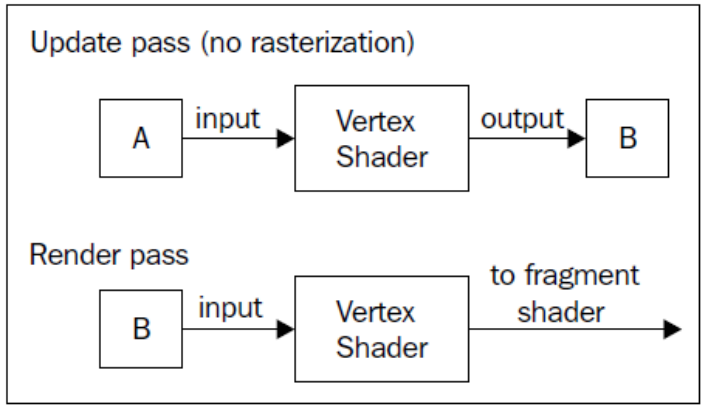
Índice

- Uno de los problemas que tiene el esquema anterior a la hora de generar el sistema de partículas es que el comportamiento de las particulas solo depende de su estado inicial y de valores constantes.
- Por ejemplo, si la aceleración no fuera constante la ecuación necesitaría conocer no solo la aceleración instantanea sino todos los valores de previos para generar la posición correcta.
- Para utilizar una aceleración instantánea es necesario actualizar la posición de la partícula en función de su posición en el instante anterior (método de Euler):

$$\mathbf{p}(t+\Delta t) = \mathbf{p}(t) + \mathbf{v}(t) \cdot \Delta t$$

$$\mathbf{v}(t+\Delta t) = \mathbf{v}(t) + \mathbf{a}(t) \cdot \Delta t$$

- Para implementar el método de Euler es necesario que el VertexShader pueda almacenar los valores de p y v en una iteración para acceder a ellos en la iteración siguiente.
- La técnica a utilizar se conoce como buffer "ping-pong" porque realiza el renderizado en dos pasadas. En la primera solo se ejecuta el VertexShader para generar el contenido del buffer. En la segunda se realiza el renderizado completo tomando como entrada el buffer generado en la primera pasada.



- Para desarollar el doble buffer se van a utilizar dos VAOs. Cada uno de ellos almacenará como atributos la posición, la velocidad, el instante inicial y la velocidad inicial.
- El primer VAO enlazará con los buffers A de la posición, velocidad e instante inicial. El segundo VAO enlazará con los buffers B de la posición, velocidad e instante inicial. Ambos VAOs enlazarán con el mismo buffer que almacena las velocidades iniciales constantes.
- A continuación se definen Transform Feedback Objects (TFO) que describen los buffers a utilizar como salida y entrada del VertexShader.

- Código

```
GLuint feedback[2];   // Transform feedback objects (TFOs)
GLuint posBuf[2];     // Position VBOs  (A and B)
GLuint velBuf[2];     // Velocity VBOs (A and B)
GLuint startTime[2];  // Start time VBOs (A and B)
GLuint initVelBuf;    // Init velocity VBO
Gluint particleArray[2];  // Particles VAOs

// Create and allocate buffers A and B for posBuf,
// velBuf and startTime
…

// Create and allocate buffer initVelBuf
…

// Create the vertex arrays
…
```

- Código

```
…

// Setup the feedback objects
glGenTransformFeedbacks(2, feedback);

// Transform feedback 0
glBindTransformFeedback(GL_TRANSFORM_FEEDBACK, feedback[0]);
glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER,0,posBuf[0]);
glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER,1,velBuf[0]);
glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER,2,startTime[0]);

// Transform feedback 1
glBindTransformFeedback(GL_TRANSFORM_FEEDBACK, feedback[1]);
glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER,0,posBuf[1]);
glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER,1,velBuf[1]);
glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER,2,startTime[1]);
```

- Antes de linkar el programa y después de compilar los shaders es necesario definir la relación entre las salidas de los shaders y los buffers que forman los FBO. Para ello se utiliza el comando *glTransformFeedbackVaryings()*.
- Durante el renderizado hay que inicializar el proceso de *transform feedback*. Normalmente se inhabilita la rasterización para que solamente se ejecute el VertexShader. Para inicializar el proceso de transform feedback se utiliza el comando *glBeginTransformFeedback()*.
- Cuando se lancen las primitivas de dibujo las salidas del VertexShader se almacenarán en los buffers indicados. Para finalizar el proceso de transform feedback se utiliza el comando *glEndTransformFeedback()*.

- VertexShader

```
subroutine void RenderPassType();
subroutine uniform RenderPassTypeRenderPass;

layout(location=0) in vec3 VertexPosition;
layout(location=1) in vec3 VertexVelocity;
layout(location=2) in float VertexStartTime;
layout(location=3) in vec3 VertexInitialVelocity;

out vec3 Position;   // To transform feedback
out vec3 Velocity;   // To transform feedback
out float StartTime; // To transform feedback
out float Transp;    // To fragment shader

uniform float Time;
uniform float H;
uniform vec3 Accel;
uniform float ParticleLifetime;
uniform mat4 MVP;
…
```

- VertexShader (sigue)

```
…

subroutine (RenderPassType)
void update() {
  Position = VertexPosition;
  Velocity = VertexVelocity;
  StartTime = VertexStartTime;
  if( Time >= StartTime ) {
    float age = Time - StartTime;
    if( age >ParticleLifetime ) {
      Position = vec3(0.0);
      Velocity = VertexInitialVelocity;
      StartTime = Time;
    } else {
    Position += Velocity * H;
    Velocity += Accel * H;
  }
}
…
```

- VertexShader (sigue)

```
…

subroutine (RenderPassType)
void render()
{
  float age = Time - VertexStartTime;
  Transp = 1.0 - age / ParticleLifetime;
  gl_Position = MVP * vec4(VertexPosition, 1.0);
}

void main()
{
  // This will call either render() or update()
  RenderPass();
}
```

- Despues de compilar los shaders y antes de linkar el programa hay que introducir el siguiente código:

```
const char * outputNames[] = { "Position", "Velocity",
                               "StartTime" };
glTransformFeedbackVaryings(progHandle, 3, outputNames,
                            GL_SEPARATE_ATTRIBS);
```

- Para lanzar la primera pasada se ejecuta lo siguiente:

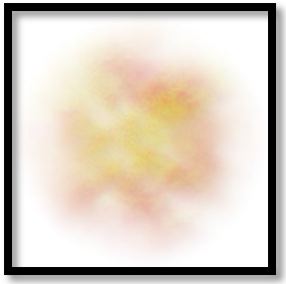
```
glUniformSubroutinesuiv(GL_VERTEX_SHADER, 1, &updateSub);
glEnable(GL_RASTERIZER_DISCARD);
glBindTransformFeedback(GL_TRANSFORM_FEEDBACK,
                        feedback[drawBuf]);

// Draw points from input buffer with transform feedback
glBeginTransformFeedback(GL_POINTS);
glBindVertexArray(particleArray[1-drawBuf]);
glDrawArrays(GL_POINTS, 0, nParticles);
glEndTransformFeedback();
```

- Para lanzar la segunda pasada se ejecuta lo siguiente:

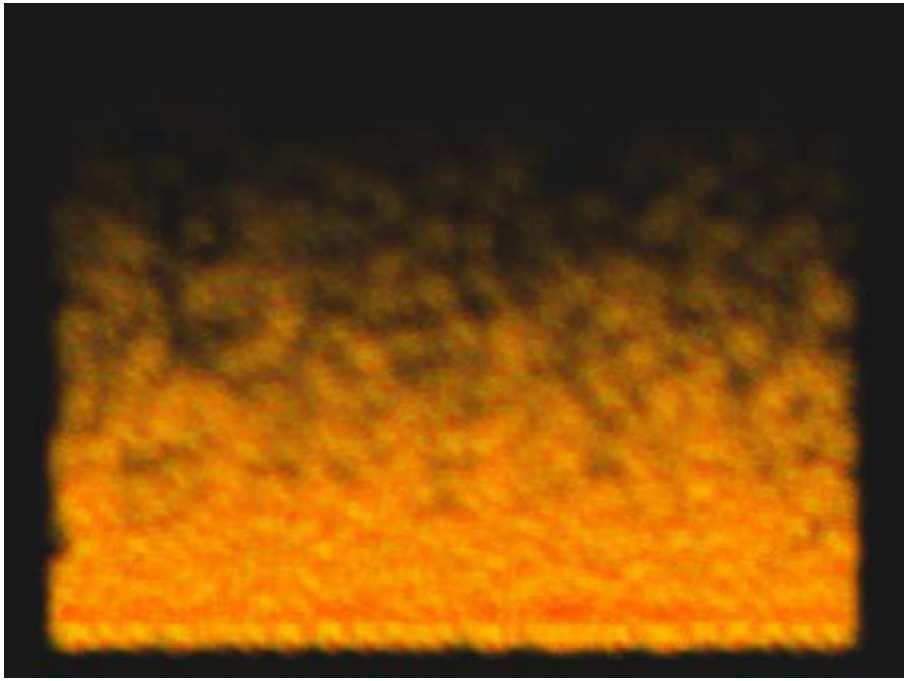
```
glDisable(GL_RASTERIZER_DISCARD);
glUniformSubroutinesuiv(GL_VERTEX_SHADER, 1, &renderSub);
glClear( GL_COLOR_BUFFER_BIT );

// Initialize uniforms for transform matrices if needed
…

// Draw the sprites from the feedback buffer
glBindVertexArray(particleArray[drawBuf]);
glDrawTransformFeedback(GL_POINTS, feedback[drawBuf]);

// Swap buffers
drawBuf = 1 - drawBuf;
```

Índice

- A partir del ejemplo anterior es fácil simular un efecto de fuego.
- En este caso la posición incial de las partículas no es la misma para todas, sino que se distribuyen entre la base de la figura.
- Las partículas de fuego no caen por gravedad sino que tienden a subir más hasta desvanecerse. Para eso se utiliza una pequeña aceleración positiva.
- Se añade a los puntos una textura formada por tonos rojos y naranjas y mucha transparencia.

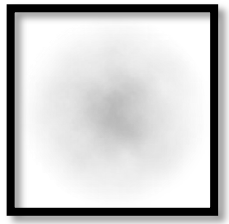


- Ejemplo de imagen generada



Índice

- Para simular el efecto de humo se parte también del esquema básico de la fuente de partículas.
- Para conseguir difuminar la partícula de humo se aumenta con el tiempo el tamaño del punto.
- Para que cada partícula pueda tener un tamaño de punto diferente es necesario activar la opción GL_PROGRAM_POINT_SIZE. A partir de ahí se puede modificar la salida gl_PointSize en el VertexShader.
- Para representar la partícula de humo se utiliza una textura en tonos grises y mucha transparencia:



- Ejemplo de imagen generada