

## Tarea 3 CC7515

Erick Sierra Baeza

DCC UChile

# Introduccion

El objetivo es implementar un visualizador de shaders en C++/GLFW que resuelva un problema específico, como la implementación de visualización de objetos.

La simulación usa CUDA para calcular fuerzas gravitacionales en paralelo y actualiza las posiciones directamente en un VBO compartido con OpenGL mediante interoperabilidad CUDA-OpenGL.

El renderizado en tiempo real se hace con shaders GLSL que aplican iluminación Phong y texturas a esferas instanciadas.

La cámara en primera persona se gestiona con matrices y vectores de GLM, y se integra con GLFW para manejo de ventanas e inputs. La interfaz interactiva permite ajustar parámetros físicos al vuelo.

# Generación de cuerpos

Se define la clase Body como:

```
class Body {  
public:  
    glm::vec3 posVec; // vector de posición (x,y,z)  
    glm::vec3 velVec; // vector de velocidad (x,y,z)  
    bool special = false; // ¿es especial?  
};
```

Luego, se crea un arreglo de tamaño  $2 * \text{DEFAULT\_N\_NUMBER}$  para manejar  $\text{DEFAULT\_N\_NUMBER}$  partículas comunes y partículas especiales, respectivamente. La constante  $\text{DEFAULT\_N\_NUMBER}$  se define en `Main.cpp` con un valor igual a 4096.

El método `void generateRandomBodies(Body* bodies, int n, int n_specials)` guarda en el arreglo `bodies` `n` partículas normales y `n_specials` partículas especiales.

# Cálculo de la interacción gravitacional

Los métodos `simulateNBodyCPU(...)` y `simulateNBodyCUDA(...)` realizan el cálculo de la interacción gravitacional entre las  $n$  partículas del arreglo `bodies` de manera secuencial y paralela, respectivamente.

Los punteros `float* mass` y `float* special_mass` apuntan a la masa de las partículas comunes y especiales, respectivamente.

En general, ambos algoritmos calculan para cada partícula  $b_i$  del arreglo `bodies`, el valor de la fuerza gravitacional con cada otra partícula  $b_j$ . La fuerza neta sobre  $b_i$  corresponderá a la suma vectorial de las fuerzas ejercidas por cada partícula del arreglo.

El cálculo de  $F$  representa la ecuación vectorial de la ley de gravitación universal.

$$\vec{F}_{21} = -G \cdot \frac{m_1 \cdot m_2}{|\vec{d}_{21}|^3} \vec{d}_{21}$$

# Renderizado

```
// creación de ventana
GLFWwindow* window = glfwCreateWindow(...);
// activar eventos de teclado
glfwSetKeyCallback(window, key_callback);
// Generar shaders, VAO, VBO y EBO y enlazarlo a los vertices
// ...
// Cargar texturas
Texture brickTex(texture_path.c_str(), GL_TEXTURE_2D, GL_TEXTURE_2D);
brickTex.texUnit(shaderProgram, "tex0", 0);
// render loop
while (!glfwWindowShouldClose(window)){
    // Dibujar esferas
    int n = numBodies + specialBodies;
    drawSpheres(bodies, shaderProgram, n);
}
```

# Renderizado

El método `drawSpheres(bodies, shaderProgram, n)` dibuja en pantalla `n` esferas del arreglo `bodies` en su respectiva posición utilizando el *vertex shader* y el *fragment shader*.

```
void drawSpheres(bodies, shaderProgram, N) {  
    // para cada partícula  
    for (int i = 0; i < N; ++i) {  
        // lee la posición  
        glm::vec3 newPos = bodies[i].posVec;  
        // inicializar render de la partícula en el origen  
        glm::mat4 model = glm::mat4(1.0f);  
        // trasladar a la posición leída  
        model = glm::translate(model, newPos);  
        // dibujar en pantalla  
        glDrawElements(GL_TRIANGLES, ...);  
    }  
}
```

# Shaders

Se utilizan vertex shader y fragment shader para las partículas y para la fuente de iluminación.

## Iluminación

### Vertex shader

Se define el layout como un vector de tres dimensiones que representan la posición. También se lee el modelo y la matriz de la cámara como uniform. La salida `gl_Position` se calcula como:

```
gl_Position = camMatrix * model * vec4(aPos, 1.0f);
```

### Fragment shader

La salida corresponde al color del vértice, se lee como uniform el color de la iluminación `lightColor`. La salida corresponde a `lightColor`.

```
out vec4 FragColor;  
FragColor = lightColor;
```

# Shaders - Partículas

## Vertex shader

### Pipeline de Transformación

Vértices → Transformación → Interpolación → Fragment Shader

### Entradas y Proceso

- **Atributos:** Posición, Color, Posición Textura, Normales
- **Uniforms:** Matriz cámara (camMatrix) + Matriz modelo (model)
- **Transformación:**  $gl\_Position = camMatrix * model * vertex$

### Salidas

- Posición mundial (crntPos)
- Color, coordenadas de textura y normales



# Shaders - Partículas

## Fragment shader

### Modelo de Iluminación Phong

Componente	Fórmula	Efecto
<b>Ambiente</b>	$0.20f$	Luz base constante
<b>Difusa</b>	$\max(\text{dot}(\text{normal}, \text{lightDir}), 0)$	Iluminación direccional
<b>Especular</b>	$\text{pow}(\text{dot}(\text{viewDir}, \text{reflectDir}), 8)$	Reflejos brillantes

### Resultado Final

```
FragColor = texture(tex0, texCoord) * lightColor *  
            (ambient + diffuse + specular)
```

# Interoperabilidad

```
cudaGraphicsResource_t cudaVB0;  
// establecer conexión entre OpenGL y CUDA  
cudaGraphicsGLRegisterBuffer(&cudaVB0, VB01.ID, ...);  
// dentro del render loop  
Body* devicePtr;  
size_t size;  
// mapear hacia un puntero CUDA  
cudaGraphicsMapResources(1, &cudaVB0);  
// obtener puntero mapeado a OpenGL  
cudaGraphicsResourceGetMappedPointer(..., cudaVB0);  
simulateNBodyCUDA(...);  
// Liberar recursos  
cudaGraphicsUnmapResources(1, &cudaVB0);
```

Fuente: <https://docs.nvidia.com/cuda/cuda-runtime-api>