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 * DEFAULT_N_NUMBER` para manejar `DEFAULT_N_NUMBER` partículas comunes y partículas especiales, respectivamente. La constante `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 `bi` del arreglo bodies, el valor de la fuerza gravitacional con cada otra partícula `bj`. La fuerza neta sobre `bi` 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.

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

Algoritmo secuencial

```
for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
        if (i == j) continue;
        // dx, dy, dz = distancia entre bi y bj
        // distancia euclidiana
        distSqr = dx * dx + dy * dy + dz * dz + NEAR_ZERO;
        // recíproco de la distancia
        invDist = rsqrtf(distSqr);
        float bi mass = (bi.special ? *special_mass : *mass);
        float bj mass = (bj.special ? *special_mass : *mass);
        // cálculo de F
```

Algoritmo paralelo

Se define un kernel que realiza el cálculo de la fuerza neta sobre una partícula utilizando la misma idea del algoritmo secuencial. Utilizando CUDA, primero se copian los datos al dispositivo. Luego se lanza el kernel y se transfieren los datos de vuelta.

```
extern "C" __global__ void updateBodies(...) {
    // cada thread maneja una partícula
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    // cálculo de F ...
}
cudaDeviceSynchronize();
// recuperar las posiciones y velocidades actualizadas
cudaMemcpy(h_bodies, d_bodies, size, cudaMemcpyDeviceToHost);
// liberar memoria
cudaFree(d_bodies);
cuCtxDestroy(context);
```

Renderizado

```
int main()
GLFWwindow* window = glfwCreateWindow(width, height, "CC7515 T
glfwSetKeyCallback(window, key_callback);
    // Generates Shader object using shaders default.vert and
    // Generates Vertex Array Object and binds it
    // Generates Vertex Buffer Object and links it to vertices
    // Generates Element Buffer Object and links it to indices
    // Links VBO attributes such as coordinates and colors to
    Texture brickTex(texture_path.c_str(), GL_TEXTURE_2D, GL_T
    brickTex.texUnit(shaderProgram, "tex0", 0);
while (!glfwWindowShouldClose(window))
```