Implementazione di un Path Tracer in Java utilizzando la libreria LWJGL per l'integrazione di OpenGL

Leonardo Notari

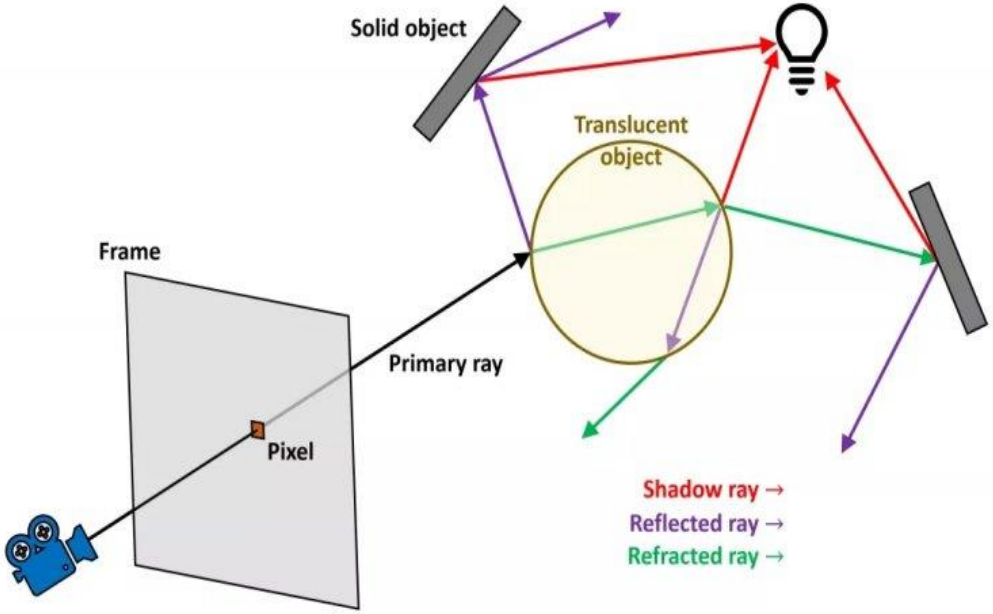
16/05/2025

Obiettivo dell' elaborato

L'obiettivo dell'elaborato è implementare un **path tracer** integrando nella pipeline di **OpenGL** il **compute shader**, un tipo speciale di shader utilizzato nelle **GPU** per eseguire calcoli generici sfruttando il parallelismo delle schede video. Verrà poi valutata l'immagine risultante in base al numero di campioni per pixel.

Introduzione: Path Tracer

La computer grafica realistica si basa su algoritmi che simulano il comportamento della luce nel mondo reale. Tra questi, uno dei più accurati è il path tracing, un metodo di rendering che segue i percorsi della luce tracciando i raggi che partono dalla telecamera e interagiscono con l'ambiente virtuale.

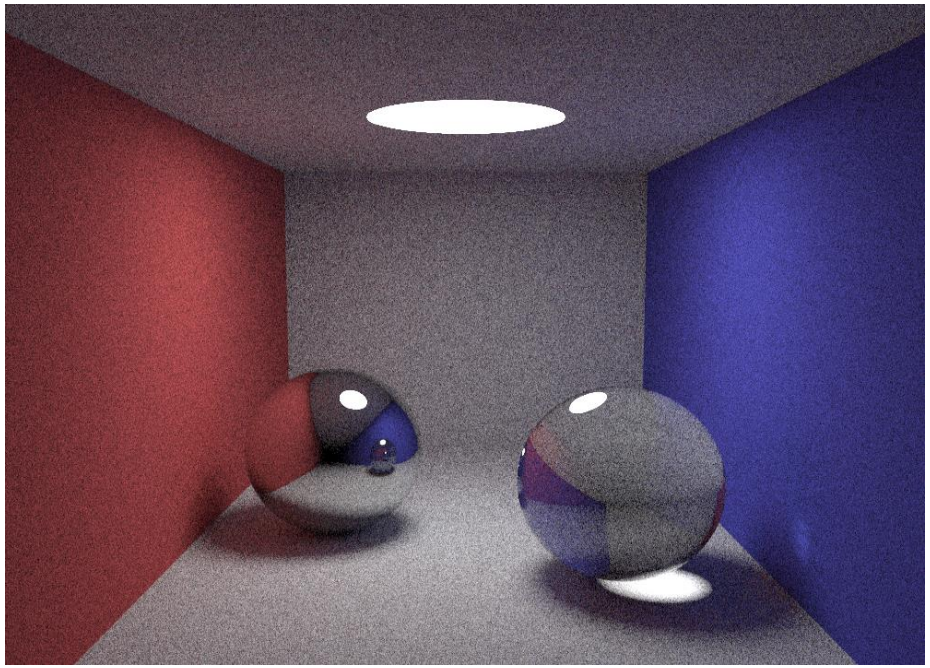


Introduzione: Libreria LWJGL

In questo progetto, è stato implementato un path tracer in **Java**, sfruttando **LWJGL** (**LightWeight Java Game Library**), una libreria open source che fornisce accesso nativo a Open GL consentendo di scrivere applicazioni in Java, fornendo binding diretti alle API native. In particolare, l'uso di OpenGL attraverso LWJGL ha permesso di gestire rendering, shader e buffer grafici in maniera efficiente.

La scena

La scena rappresentata è una stanza con pareti di colori diversi, illuminata da una luce che proviene dal soffitto. All'interno della stanza ci sono due sfere, ciascuna di materiale diverso che interagisce in modo differente con i raggi luminosi.



256 campioni per pixel

La scena

Per costruire la scena sono state utilizzate delle sfere con raggio estremamente grande per le pareti, semplificando il test per verificare le intersezioni.
I materiali che la compongono possono essere di quattro tipi a seconda di come interagiscono con la luce:

Lambertiano (o diffusivo): per le pareti, il pavimento e il soffitto.

Rifrangente: una delle sfere devia la luce, facendo cambiare direzione al raggio.

Riflettente: l'altra sfera riflette in modo speculare la luce.

Emissivo: per illuminare la stanza è stata inserita una sfera che emette luce.

Presentazione codice: Java

Per generare un'immagine con il compute shader è stata inizialmente creata e allocata una **texture** delle dimensioni della finestra.

Codice Java:

```
int outputTexture = glGenTextures();
glBindTexture(GL_TEXTURE_2D, texture);
[…]
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA32F, width, height, 0,
GL_RGBA,GL_FLOAT, 0);
```

Successivamente è stato fatto il **binding** della texture all'*image unit 0* (corrispondente al *binding point 0)* per far scrivere il colore calcolato dal compute shader direttamente dal codice **GLSL**.

Codice Java:

```
glBindImageTexture(0, outputTexture, 0, false, 0, GL_WRITE_ONLY,GL_RGBA32F);
```

Presentazione codice: Java

In modo simile sono passate anche le sfere, utilizzando però l'**SSBO (Shader Storage Buffer Object)**, una struttura dati flessibile particolarmente utile nei compute shader quando si gestiscono grandi quantità di dati, e il *binding point 1*.

Codice Java:

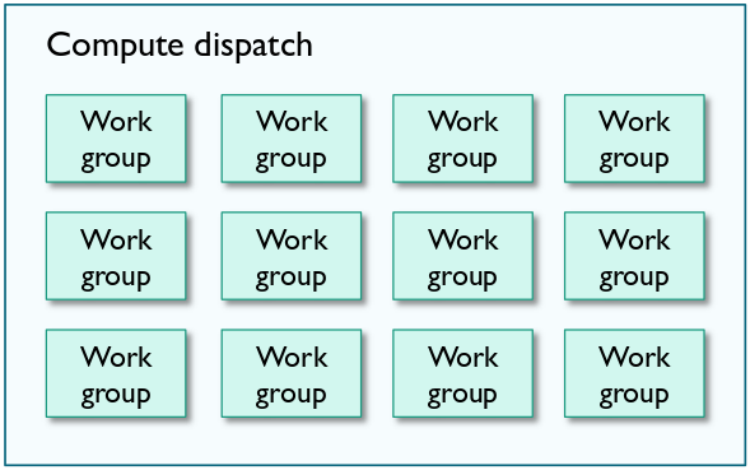
```
FloatBuffer buffer = BufferUtils.createFloatBuffer(spheres.length * 8);
for (Sphere sphere : spheres) {
  buffer.put(sphere.center.x).put(sphere.center.y).put(sphere.center.z);
  buffer.put(sphere.radius);
  buffer.put(sphere.albedo.x).put(sphere.albedo.y).put(sphere.albedo.z);
  buffer.put(sphere.materialType);
}
buffer.flip();
int ssbo = glGenBuffers();
glBindBuffer(GL_SHADER_STORAGE_BUFFER, ssbo);
glBufferData(GL_SHADER_STORAGE_BUFFER, buffer, GL_STATIC_DRAW);
glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 1, ssbo);
```

Presentazione codice: Java

Codice Java:

```
int workGroupSize = 16;
int workGroupsX = (width + workGroupSize - 1) / workGroupSize;
int workGroupsY = (height + workGroupSize - 1) / workGroupSize;
glDispatchCompute(workGroupsX, workGroupsY, 1);
glMemoryBarrier(GL_SHADER_IMAGE_ACCESS_BARRIER_BIT);
```

In questo codice si calcola il numero di **work group** che serviranno a coprire l'intera immagine. Ogni work group è un insieme di **thread** (detti *invocations*) che calcolano in parallelo il colore dei pixel in una porzione dell'immagine, in questo caso di 16×16 pixel.



Presentazione codice: Java

Infine, viene generato un rettangolo con un semplice **vertex shader**, con **vao** e **vbo** creati in precedenza, sul quale la texture viene ''incollata'' nel **fragment shader**.

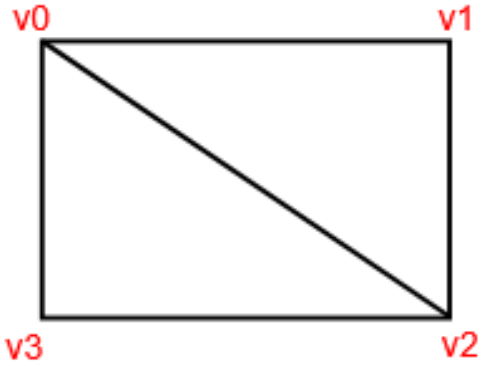
```
Codice Java:
  float[] quadVertices = { -1.0f,  1.0f, 0.0f, 0.0f, 1.0f, …}
  quadVAO = glGenVertexArrays();
  quadVBO = glGenBuffers();
  glBindVertexArray(quadVAO);
  glBindBuffer(GL_ARRAY_BUFFER, quadVBO);
  glBufferData(GL_ARRAY_BUFFER, quadVertices, GL_STATIC_DRAW);
  glEnableVertexAttribArray(0);
  glVertexAttribPointer(0,3,GL_FLOAT,false,5*Float.BYTES,0);
  glEnableVertexAttribArray(1);
  glVertexAttribPointer(1,2,GL_FLOAT,false,5*Float.BYTES,3*Float.BYTES);
```

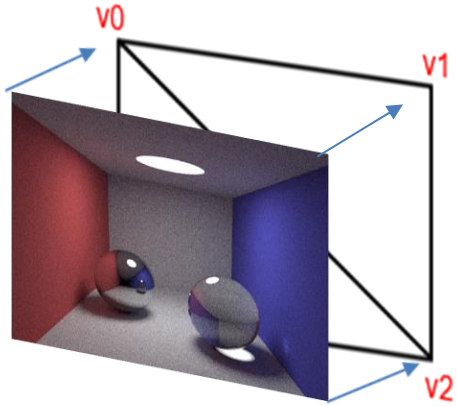
```
Codice Java:
  glUseProgram(displayShader);
  glActiveTexture(GL_TEXTURE0);
  glUniform1i(glGetUniformLocation(displayShader, "screenTexture"), 0);
  glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
  glfwSwapBuffers(window);
```

Presentazione codice: shaders

```
GLSL vertex shader:
  layout (location = 0) in vec3 aPos;
  layout (location = 1) in vec2 aTexCoords;
  out vec2 TexCoords;
  void main() {
    TexCoords = aTexCoords;
    gl_Position = vec4(aPos, 1.0);
  }

GLSL fragment shader:
  out vec4 FragColor;
  in vec2 TexCoords;
  uniform sampler2D screenTexture;
  void main() {
    vec3 color = texture(screenTexture, TexCoords).rgb;
    color = pow(color, vec3(1.0/2.2));
    FragColor = vec4(color, 1.0);
  }
```





Presentazione codice: shaders

All'inizio del compute shader, viene definita la dimensione di ciascun work group, che conterrà 16x16 thread, ognuno dei quali eseguirà il codice. La seconda riga crea il collegamento con la texture. Infine, viene letto anche il buffer contenente le informazioni delle sfere.

GLSL compute shader:

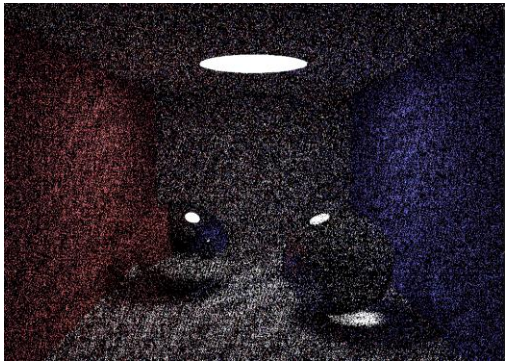
```
layout(local_size_x = 16, local_size_y = 16) in;
layout(rgba32f, binding = 0) uniform image2D img_output;
layout(std430, binding = 1) buffer Scene{ vec4 spheres[]; }
```

Presentazione codice: shaders

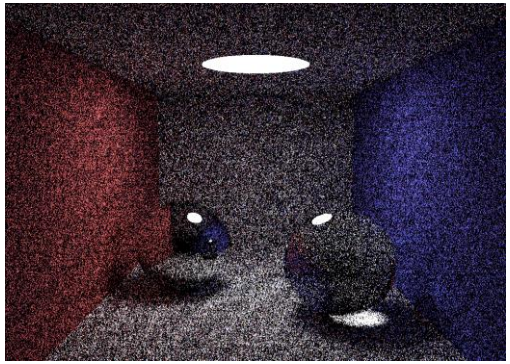
Nel main, ogni thread si occuperà di calcolare il colore di un pixel identificato dalle coordinate restituite da *ivec(gl_GlobalInvocationID.xy).* Per farlo esegue campionamenti **Monte Carlo**, media i risultati e scrive il colore finale direttamente nella texture di output tramite *imageStore*, una funzione GLSL che scrive un valore in una immagine a coordinate precise.

```
GLSL compute shader:
  void main(){
    ivec2 pixelCoords = ivec2(gl_GlobalInvocationID.xy);
    […]
    totalColor = vec3(0.0);
    for (int i = 0; i < samples; i++){
      […]
      vec3 color = calculateColor(cameraPos, rayDir);
      totalColor += color;}
    vec3 finalColor = totalColor / float(samples);
    imageStore(img_output, pixelCoords, vec4(finalColor, 1.0));
  }
```

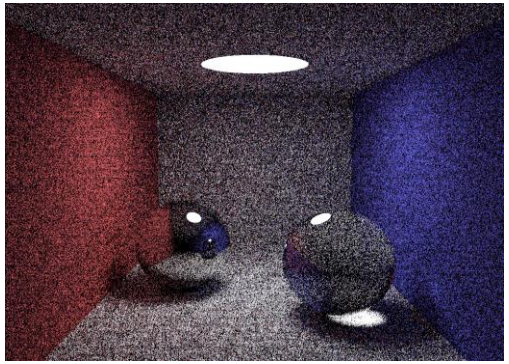
Risultati



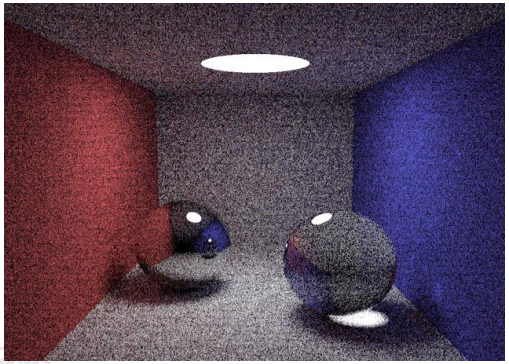
4 campioni per pixel



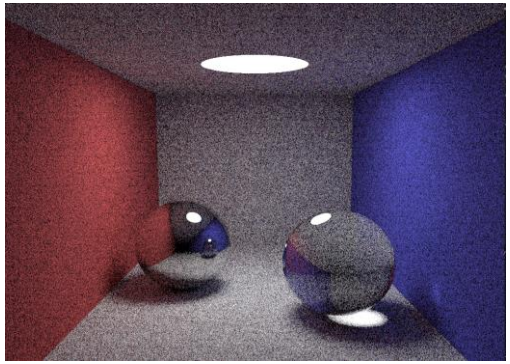
8 campioni per pixel



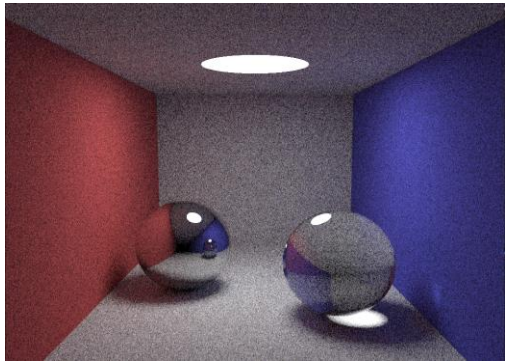
16 campioni per pixel



32 campioni per pixel

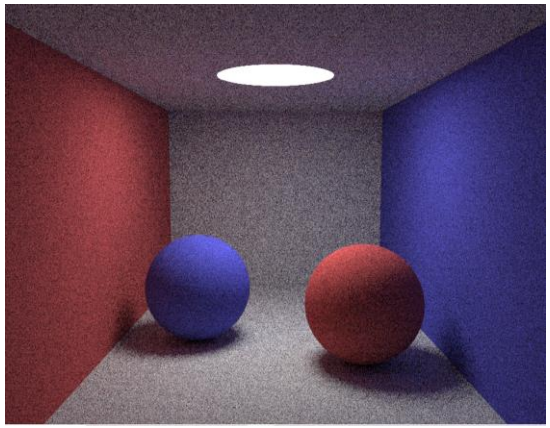


64 campioni per pixel

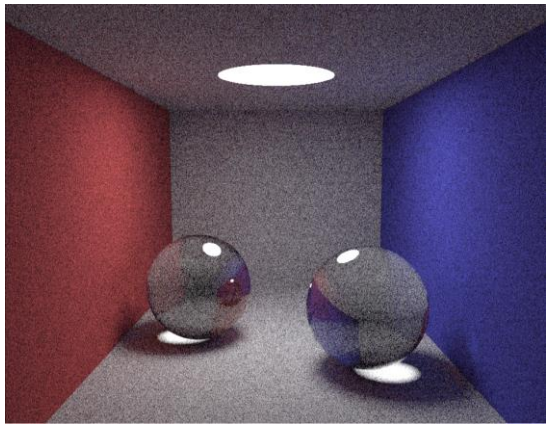


128 campioni per pixel

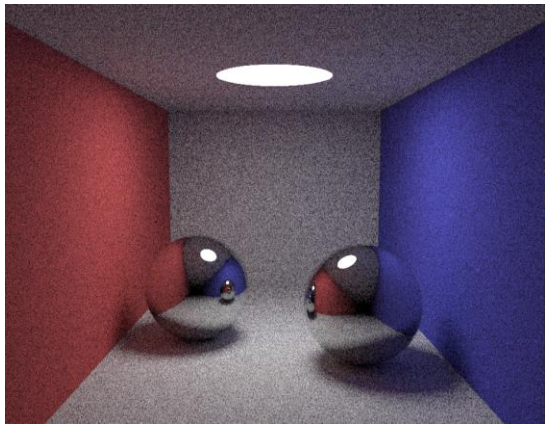
Variazioni della scena



In basso la scena rappresentata due sfere che riflettono specularmente i raggi, simulando una superficie specchiata.



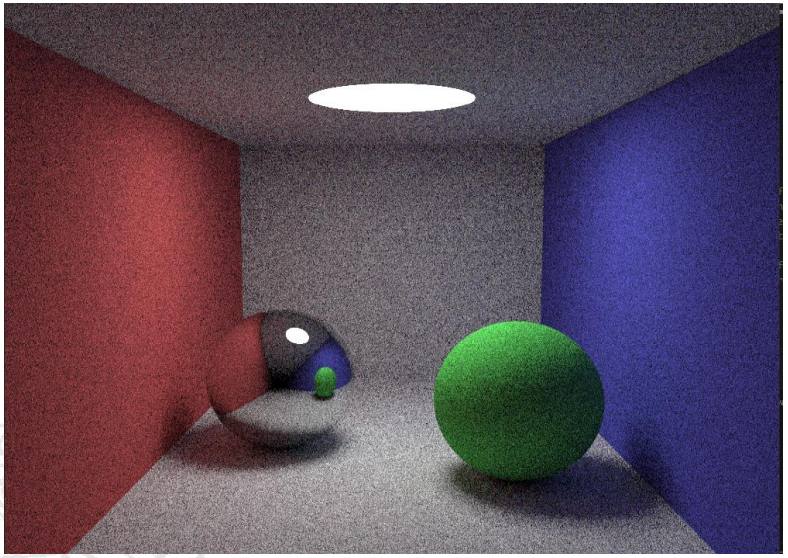
In alto due sfere di un materiale lambertiano, che disperde raggi in modo casuale favorendo direzioni vicino alla normale.

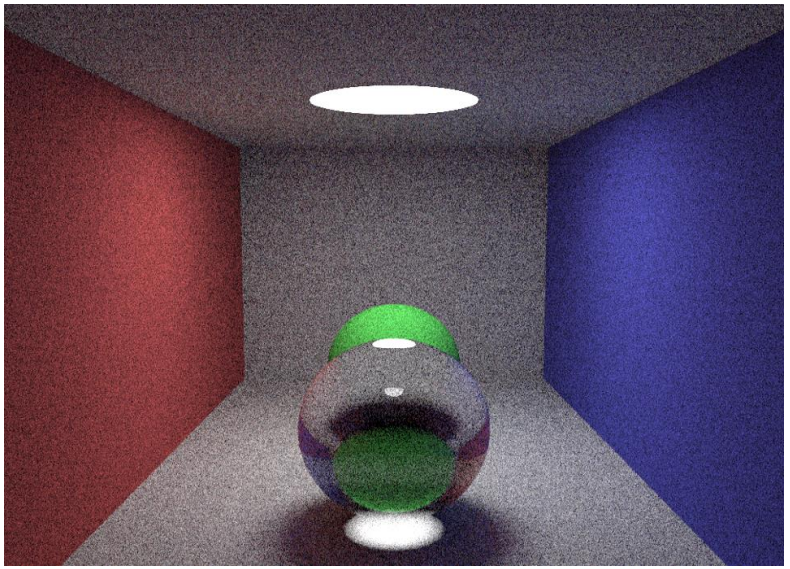


In quest'ultima entrambe le sfere simulano il vetro, suddividendo i raggi in una componente rifratta ed una riflessa.

Variazioni della scena

Nell'immagine sottostante una delle sfere riflette l'immagine dell'altra di materiale diffusivo.





Sopra, invece, le due sfere sono state posizionate una dietro l'altra ed è possibile osservare la sfera di materiale lambertiano capovolta attraverso la prima, di vetro.

Conclusioni

Il progetto ha dimostrato l'efficacia del path tracing implementato con compute shaders in Java tramite LWJGL. Aumentando il numero di campioni per pixel, il rumore visibile nell'immagine diminuisce significativamente, migliorando la qualità del rendering. Tuttavia, l'aumento dei campioni comporta un incremento della complessità computazionale, evidenziando un trade-off tra qualità e performance.