Introduction à l'Informatique Graphique

- Généralités
- Concepts d'une scène 3D
- Coordonnées généralisées
- **OpenGL et notion de Shaders**

CPU et GPU

CPU: Central Processing Unit

Mémoire associée: RAM

⇒ Tâches arbitraires séquentielles:

Calcul, branching, accès mémoires, I/O, etc.

Coeurs: 2 à 64, indépendants

RAM: 4 à 64 Go

GPU: Graphics Processing Unit

Mémoire associée: VRAM

⇒ Tâches parallèles avec opérations similaires:

Calculs vectoriels, matriciels sur tableaux de données.

Architecture: SIMD

Single Instruction, Multiple Data

Coeurs: 1000 à 16000, partagés

VRAM: 4 à 16 Go

GPU: Supercalculateur pour réaliser la même opération sur un grand nombre de données.

OpenGL

OpenGL: Open Graphics Library

Une API pour communiquer avec le GPU, orienté graphique 3D

API: Application Programming Interface = Ensemble standardisé de variables et en-tête de fonctions.

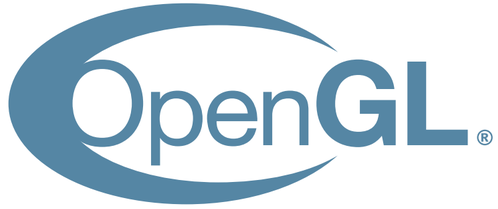
Bas niveau, Haute performance, Multi plateformes

Rem.

API $\neq$ logiciel, librairie

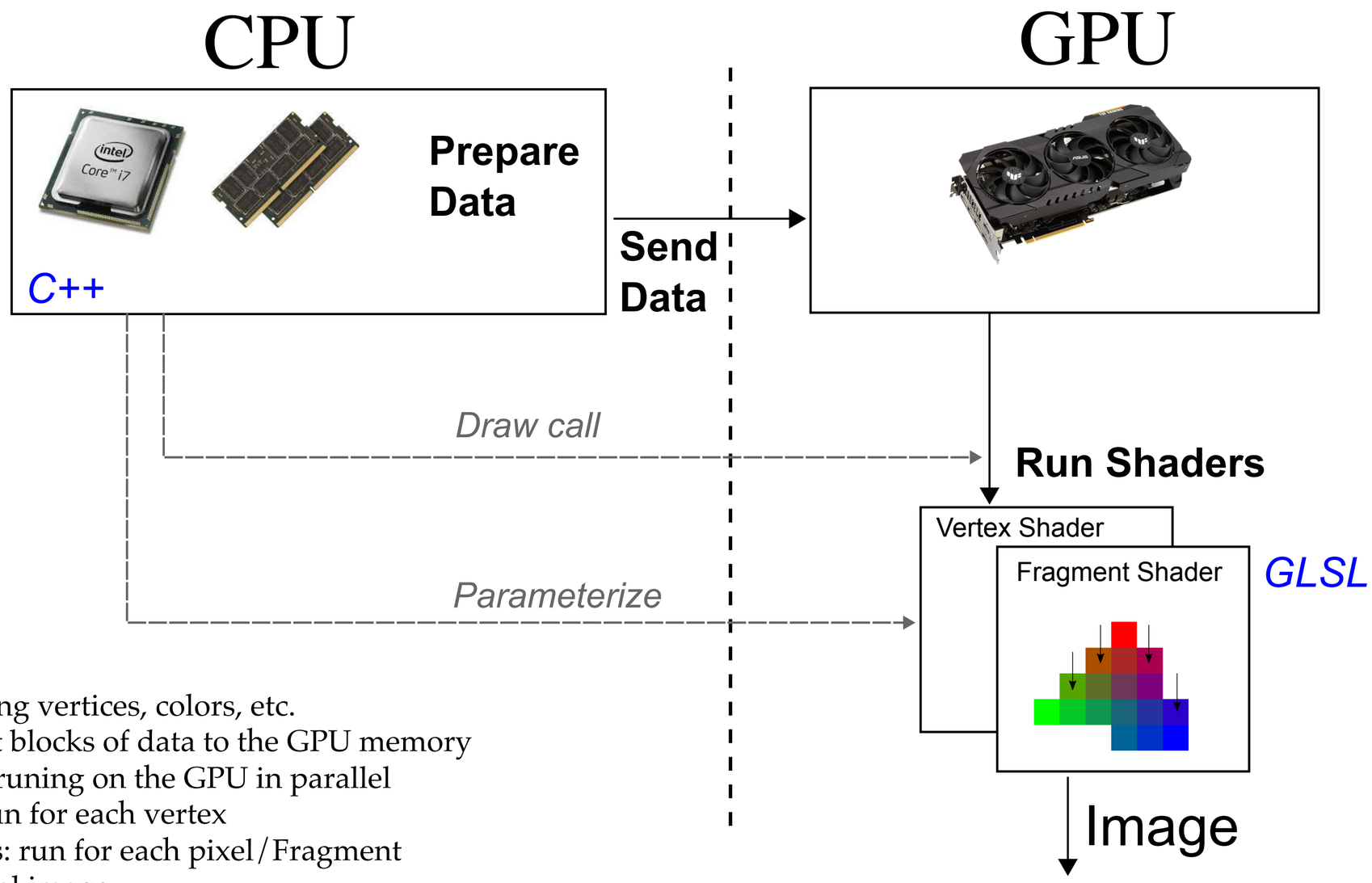
Différents systèmes/GPU ont différentes implémentations d'OpenGL

Installation dépend du driver



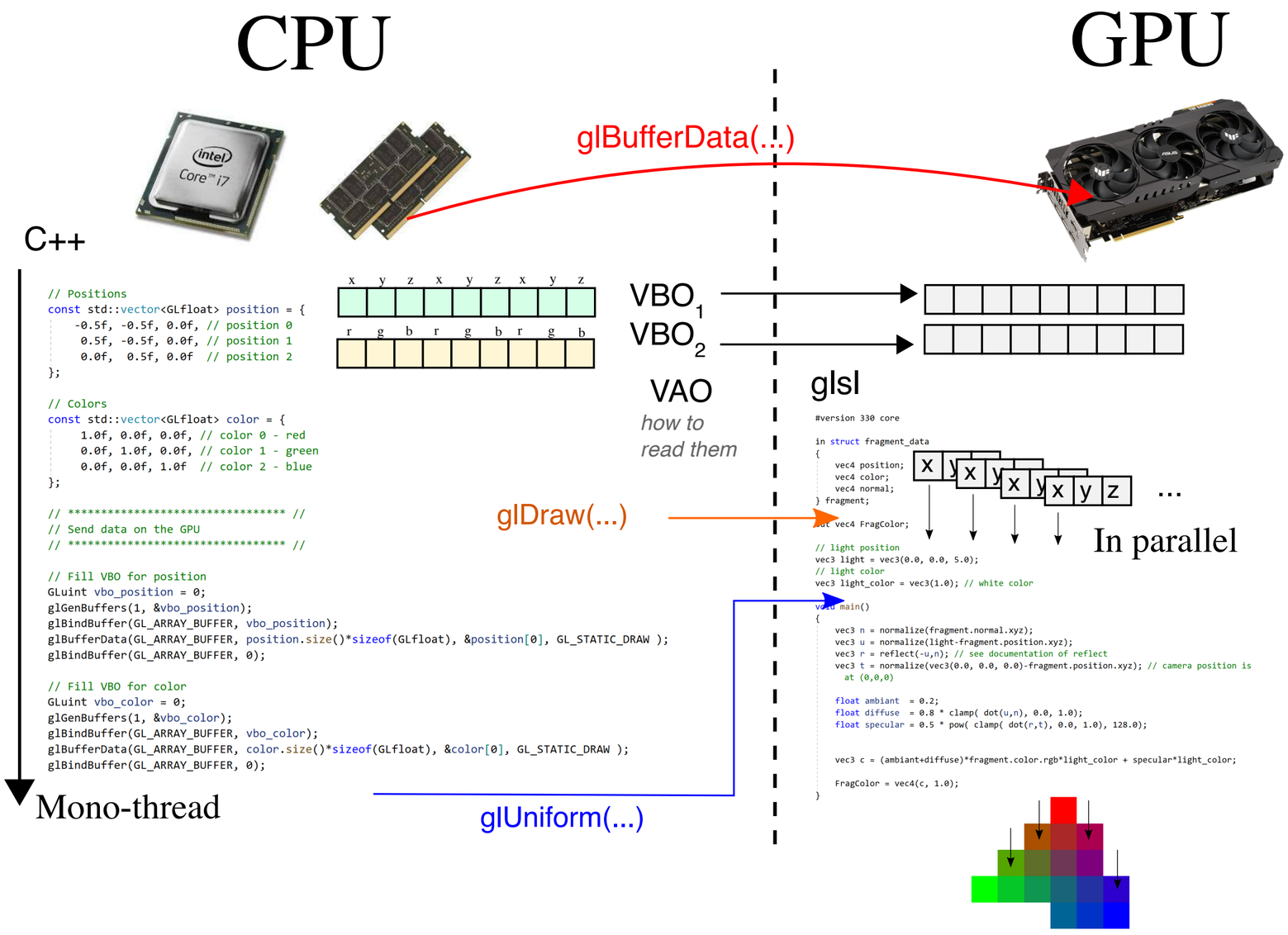
Autres APIS: Vulkan, WebGL, WebGPU, DirectX (Windows), Metal (Mac).

Communication CPU/GPU with OpenGL



1- Preparing the data:
Loading/Computing vertices, colors, etc.
2- Send data: Transfert blocks of data to the GPU memory
3- Shaders: Programs runing on the GPU in parallel
- Vertex shaders: run for each vertex
- Fragment shaders: run for each pixel/Fragment
Output: The final image

Communication CPU/GPU with OpenGL - details




```
// Positions
const std::vector<GLfloat> position = {
    -0.5f, -0.5f, 0.0f, // position 0
     0.5f, -0.5f, 0.0f, // position 1
     0.0f,  0.5f, 0.0f  // position 2
};

// Colors
const std::vector<GLfloat> color = {
     1.0f, 0.0f, 0.0f, // color 0 - red
     0.0f, 1.0f, 0.0f, // color 1 - green
     0.0f, 0.0f, 1.0f  // color 2 - blue
};

// ********************************* //
// Send data on the GPU
// ********************************* //

// Fill VBO for position
GLuint vbo_position = 0;
glGenBuffers(1, &vbo_position);
glBindBuffer(GL_ARRAY_BUFFER, vbo_position);
glBufferData(GL_ARRAY_BUFFER, position.size()*sizeof(GLfloat), &position[0], GL_STATIC_DRAW );
glBindBuffer(GL_ARRAY_BUFFER, 0);

// Fill VBO for color
GLuint vbo_color = 0;
glGenBuffers(1, &vbo_color);
glBindBuffer(GL_ARRAY_BUFFER, vbo_color);
glBufferData(GL_ARRAY_BUFFER, color.size()*sizeof(GLfloat), &color[0], GL_STATIC_DRAW );
glBindBuffer(GL_ARRAY_BUFFER, 0);
```

```
#version 330 core

in struct fragment_data
{
    vec4 position;
    vec4 color;
    vec4 normal;
} fragment;

out vec4 FragColor;

// light position
vec3 light = vec3(0.0, 0.0, 5.0);
// light color
vec3 light_color = vec3(1.0); // white color

void main()
{
    vec3 n = normalize(fragment.normal.xyz);
    vec3 u = normalize(light-fragment.position.xyz);
    vec3 r = reflect(-u,n); // see documentation of reflect
    vec3 t = normalize(vec3(0.0, 0.0, 0.0)-fragment.position.xyz); // camera position is
      at (0,0,0)

    float ambiant  = 0.2;
    float diffuse  = 0.8 * clamp( dot(u,n), 0.0, 1.0);
    float specular = 0.5 * pow( clamp( dot(r,t), 0.0, 1.0), 128.0);


    vec3 c = (ambiant+diffuse)*fragment.color.rgb*light_color + specular*light_color;

    FragColor = vec4(c, 1.0);
}
```

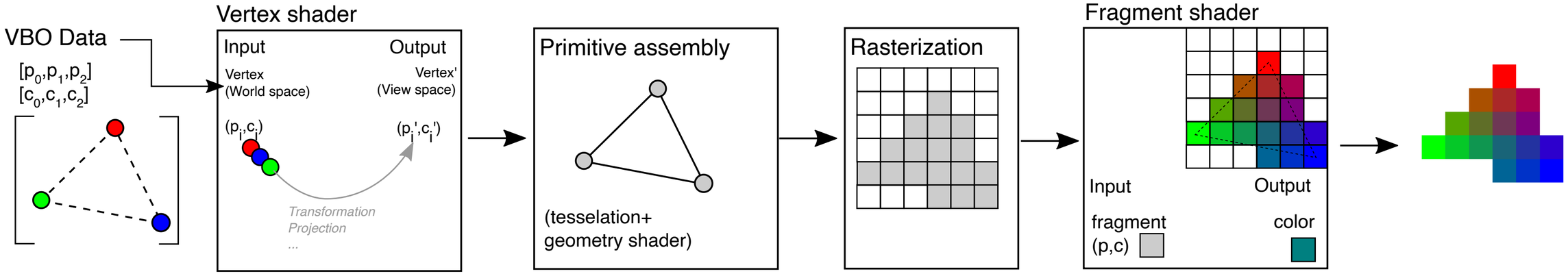
Shader

Vertex Shader

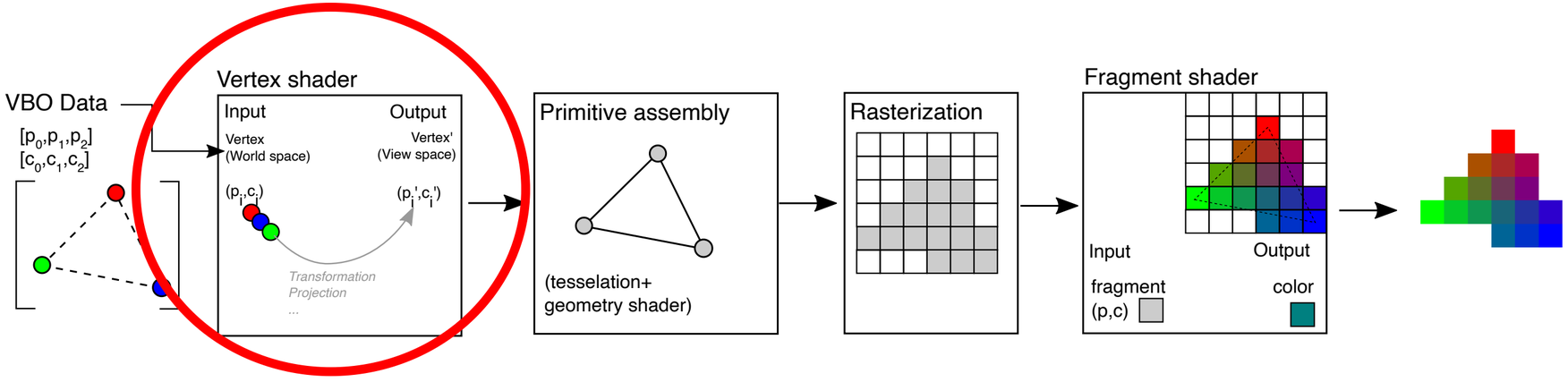
```
#version 330 code
layout(location = 0) in vec4 position;
layout(location = 1) in vec4 color;

out vec4 vertexColor;

void main()
{
    gl_Position = position;
    vertexColor = color;
}
```

Fragment Shader

```
#version 330 code
in vec4 vertexColor;
out vec4 fragColor;

void main()
{
    fragColor = vertexColor;
}
```



Transformation des sommets
Projection et perspective



Synthèse matrices - Formulation

Entrée: Coordonnées $p = (x, y, z, 1)$ dans l'espace objet

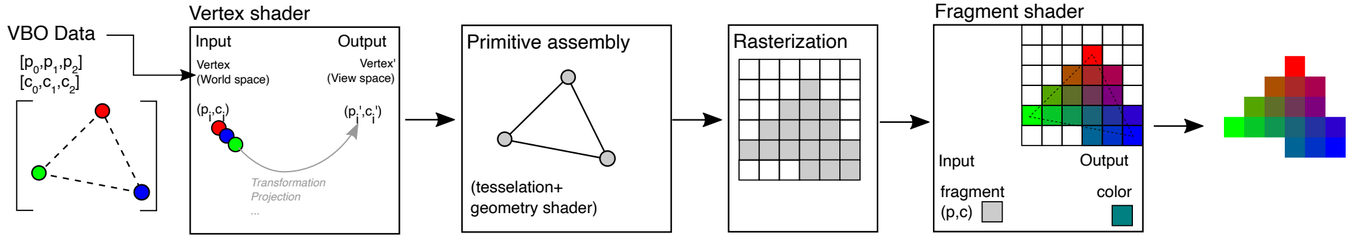
- Coordonnées dans l'espace du monde $p_{ws} = \text{Model } p$
- Coordonnées dans l'espace caméra $p_{view} = \text{View } p_{ws}$
- Coordonnées dans le NDC - Normalized Device Coordinates $p_{ndc} = \text{Proj } p_{view}$

Sortie en espace image: NDC homogénéisée $p_{out} = (x_{ndc}/w_{ndc},\ y_{ndc}/w_{ndc},\ z_{ndc}/w_{ndc},\ 1)$

Formulation globale: $\boxed{p_{ndc} = \text{Proj} \times \text{View} \times \text{Model} \ \ p}$

1ère étape du pipeline graphique: **Vertex shader**

```
#version 330 core

// vertex position in local space (x,y,z)
layout (location = 0) in vec3 vertex_position;

// uniform = parameters send from C++ code
uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main() {
    // gl_Position: Output position from Vertex Shader (NDC)
    gl_Position = projection * view * model * (position,1.0);

    // normalization automatically performed by the GPU
}
```



Synthèse matrices - Demo

$$p_{ndc} = \text{Proj} \times \text{View} \times \text{Model} \;\; p$$

Matrices: C++ / paramètres uniformes

Changement de coordonnées en C++:

- Principalement mono-thread
  - Calcul des nouvelles positions
  - Copies coordonnées sur mémoire GPU
- Possible, mais lent

```
std::vector<vec3> apply(mat4 Model, mat4 View, mat4 Projection,
                        std::vector<vec3> const& position_in) {
    std::vector<vec3> position_out;
    for (auto const& p : p_in) {
        vec4 p_homog{p, 1.0f};
        vec4 p_ndc_homog = Projection * View * Model * v_homog;
        vec3 p_ndc = vec3{v_clip} / v_clip.w;
        position_out.push_back(p_ndc);
    }
    return position_out;
}

void main_loop() {
    std::vector<vec3> new_position;
    // Update Model, View, Projection, ...

    // N-operations - monothread
    new_position = apply(Model, View, Projection, mesh.position);

    // N-copy to the GPU
    mesh_drawable.position.update(new_position);

    draw(mesh_drawable);
}
```

Matrices: C++ / paramètres uniformes

Changement de coordonnées via Uniforms:

- Simple copie de matrices
- Pas de calcul supplémentaire en C++
- Calcul des nouvelles positions en parallèle sur le GPU, surcout négligeable

```cpp
void main_loop() {
    // Update Model, View, Perspective, ...

    // Copy three 4x4 matrices
    glUniform(shader, Model);
    glUniform(shader, View);
    glUniform(shader, Projection);

    draw(mesh_drawable);
}
```

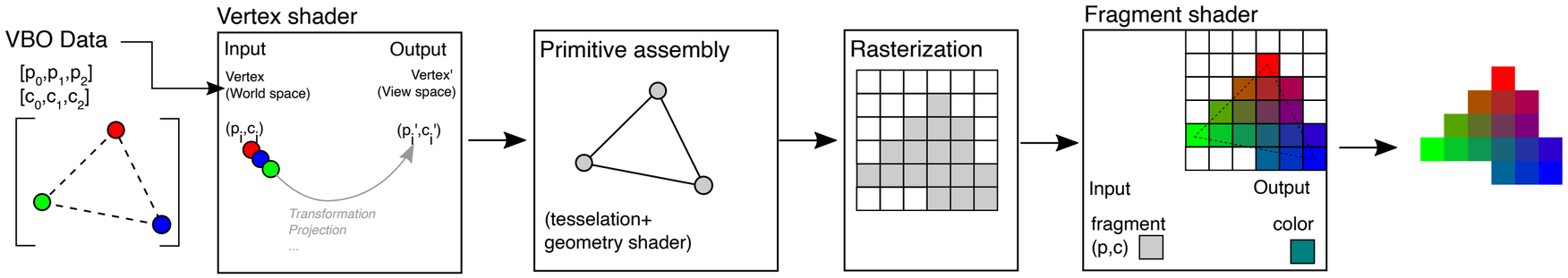
C++

```
in vec3 vertex_position;
uniform mat4 Model;
uniform mat4 View;
uniform mat4 Projection;
void main() {
    gl_Position = Projection * View * Model * (position,1.0);
}
```

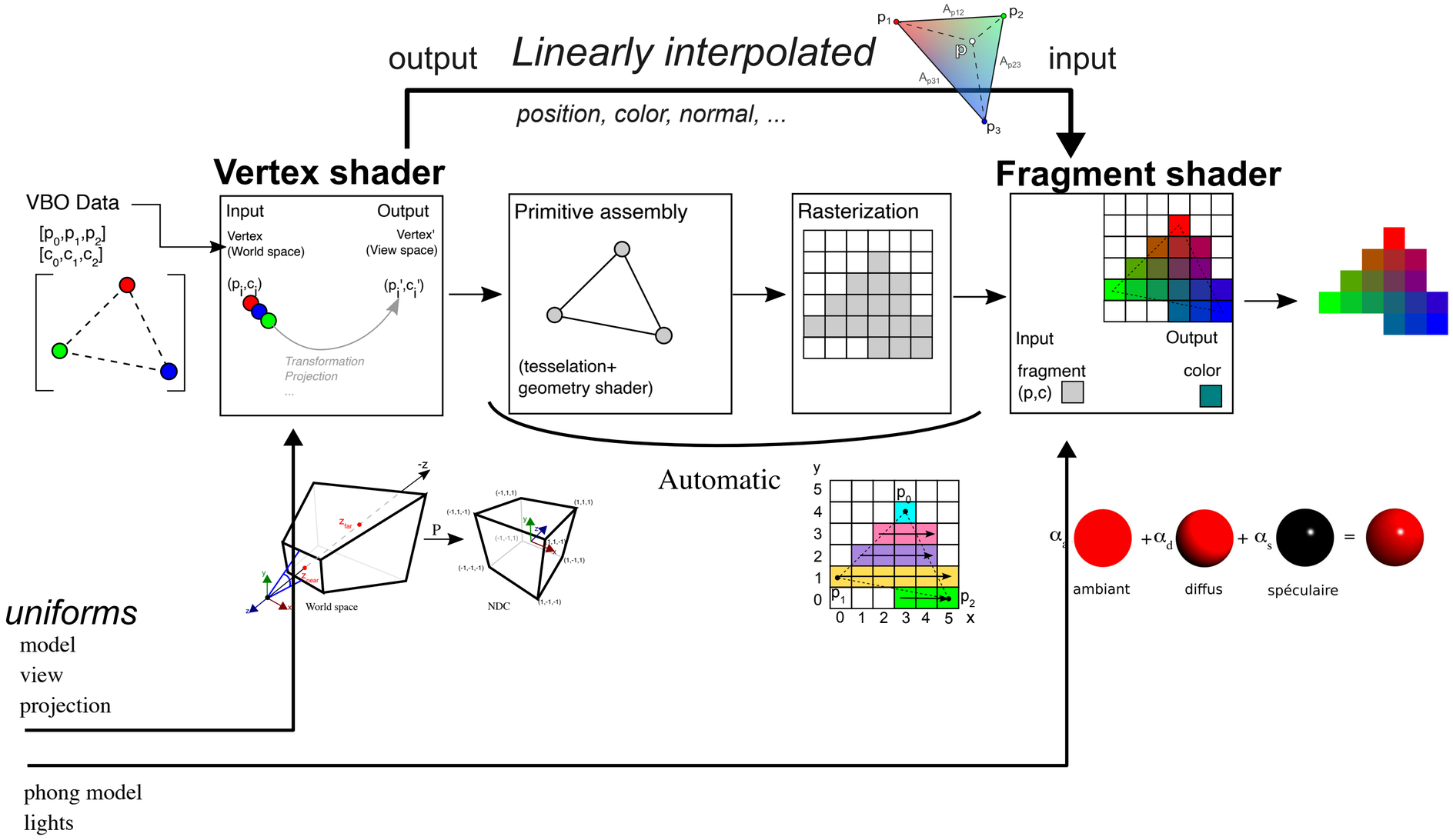
In parallel on all vertices

Vertex shader

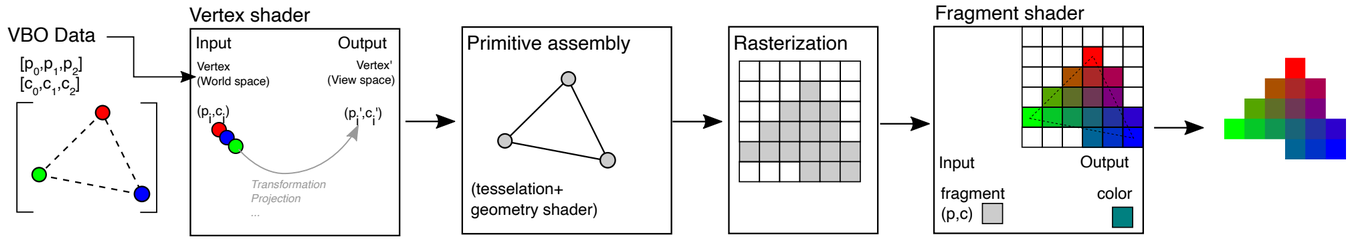
Shaders: résumé



Shaders Résumé



Shaders Résumé: Exemple de code




```glsl
layout (location = 0) in vec3 vertex_position;
layout (location = 1) in vec3 vertex_normal;
layout (location = 2) in vec3 vertex_color;

out struct fragment_data {
    vec3 position;
    vec3 normal;
    vec3 color;
} fragment;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main() {
    vec4 position = model * vec4(vertex_position, 1.0);

    mat4 modelNormal = transpose(inverse(model));
    vec4 normal = modelNormal * vec4(vertex_normal, 0.0);

    fragment.position = position.xyz;
    fragment.normal   = normal.xyz;
    fragment.color = vertex_color;

    gl_Position = projection * view * position;
}
```

```glsl
// Interpolated values on the fragment
in struct fragment_data {
    vec3 position;
    vec3 normal;
    vec3 color;
} fragment;

uniform vec3 light_position;
uniform vec3 light_color;

layout(location=0) out vec4 FragColor;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main() {

    vec3 N = normalize(fragment.normal);
    vec3 L = normalize(light_position-fragment.position);

    float diffuse_magnitude = max(dot(N,L),0.0);
    // ...

    vec3 c = a_ambient * fragment.color*light_color;
    c = c + a_diffuse*diffuse_magnitude*fragment.color*light_color;
    c = c + a_specular*specular_magnitude*light_color;

    FragColor = vec4(c,1.0);
}
```