

TP5 - Shaders

Concepts and Practice

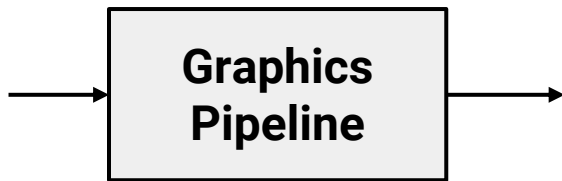
From object definition to rendering

Objects are defined by **vertices**, **indices**, **normals** and **texture coordinates**

Let's see how this data is used to render objects in the scene

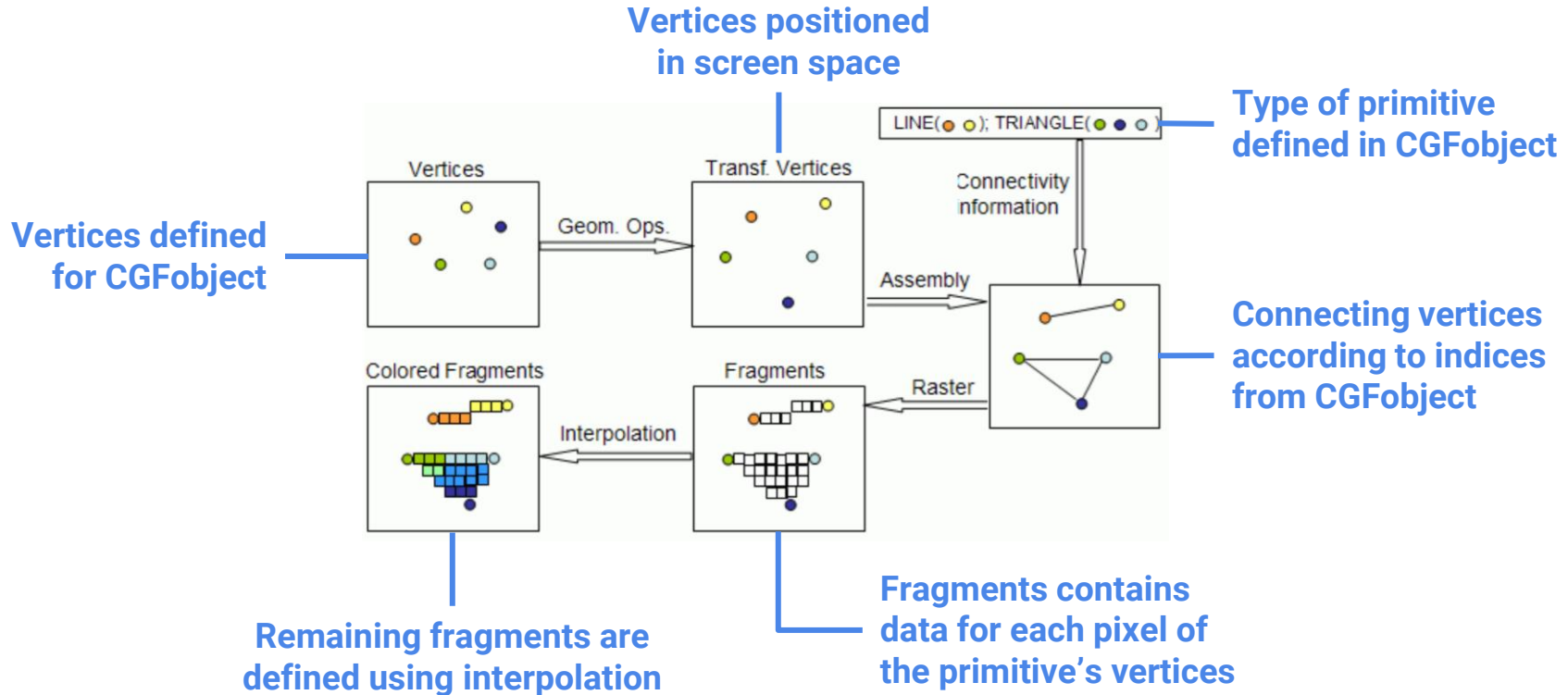
```
...  
initBuffers(){  
    this.vertices = [...]  
    this.indices = [...]  
    this.normals = [...]  
    this.texCoords = [...]  
}
```

Object definition using
CGFobject class



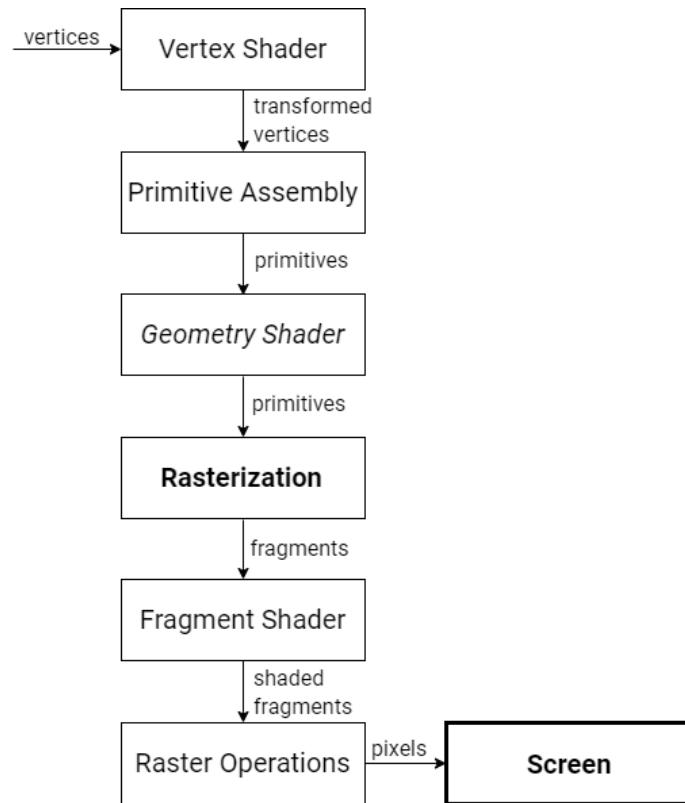
Rendered scene in screen

Graphics Pipeline - Visualization



Graphics Pipeline

- Inputs
- **Vertex shading**
- Primitive assembly
- Geometry shading
- Projection and rasterization
- **Fragment shading**
- Raster operations
- Output to screen



Shaders

Small programs receive and manipulate data on the 3D scene:

- **Vertex shaders** - Manipulate and define properties **for each vertex**
- **Fragment shaders** - Manipulate and define properties **for each fragment**

Custom data may be passed to the shaders from the application

Data may be passed **from vertex to fragment shader** (not inversely)

Shaders in WebGL/WebCGF

In **WebGL**, shaders may be loaded as strings and compiled in real time

To apply shaders to a scene using **WebCGF**, these are the general steps:

- 1 **Create** the shader files
- 2 Load the created files to a ***CGFshader*** class object
- 3 Set *CGFshader* object as the **scene's active shader**
- 4 **Pass values** from application to shaders (optional)

1 Creating Shaders - Structure

Shader may be defined in `.vert` or `.frag` files, for vertex or fragment shaders

A shader program commonly contains:

- List of **input/output** and **uniform** variables
- A ***main()* function**, where input data is processed and output is returned

The *main()* function runs for **each vertex** or for **each fragment**

1 Creating Shaders - Vertex Shader

A vertex shader receives **input** relative to:

- each **vertex** (position, normal, texture coordinate)
- lights, materials, camera (for illumination model and other functionalities)
- **custom data** provided from the application

A vertex shader creates as **output**:

- the **calculated position** for each vertex
- data to be passed to the **fragment shader**

1 Creating Shaders - Vertex Shader Example

```
attribute vec3 aVertexPosition;  
attribute vec3 aVertexNormal;  
attribute vec2 aTextureCoord;
```

```
uniform mat4 uMVMatrix;  
uniform mat4 uPMatrix;  
uniform mat4 uNMatrix;
```

Input variables

...

```
void main() {
```

Main function, processes each vertex

...

```
    gl_Position = uPMatrix * uMVMatrix * vec4(aVertexPosition, 1.0);
```

Output vertex position

```
}
```

1 Creating Shaders - Variables

```
attribute vec3 aVertexPosition
```

Qualifier

Data Type Qualifiers

- `uniform` - read-only global input, from WebGL or application
- `attribute` - read-only per-vertex input to vertex shader
- `varying`
 - writable output (vertex shader)
 - read-only input (fragment shader)
- `const` - read-only compile-time constant

1 Creating Shaders - Variables

```
attribute vec3 aVertexPosition
```

Type

Data Type

- **Scalars** - bool, int, float, ...
- **Vectors** - vec2, vec3, vec4, ...
- **Matrices** - mat2, mat3, mat4, ...
- **Textures** sampler1D, sampler2D, sampler3D, ...
- And others

1 Creating Shaders - Variables in Vertex Shader

```
attribute vec3 aVertexPosition;
attribute vec3 aVertexNormal;
attribute vec2 aTextureCoord;
uniform mat4 uMVMatrix;
uniform mat4 uPMatrix;
uniform mat4 uNMatrix;
...

void main() {
    ...
    gl_Position = uPMatrix * uMVMatrix * vec4(aVertexPosition, 1.0);
}
```

— position of **each vertex** (attribute) as **3D vector** (vec3)

— **global** (uniform) model-view **4x4 matrix** (mat4)

1 Creating Shaders - Vertex Shader main() function

```
void main() {  
    ...  
    gl_Position = uPMatrix * uMVMMatrix * vec4(aVertexPosition, 1.0);  
}
```

Output vertex position

Projection Matrix

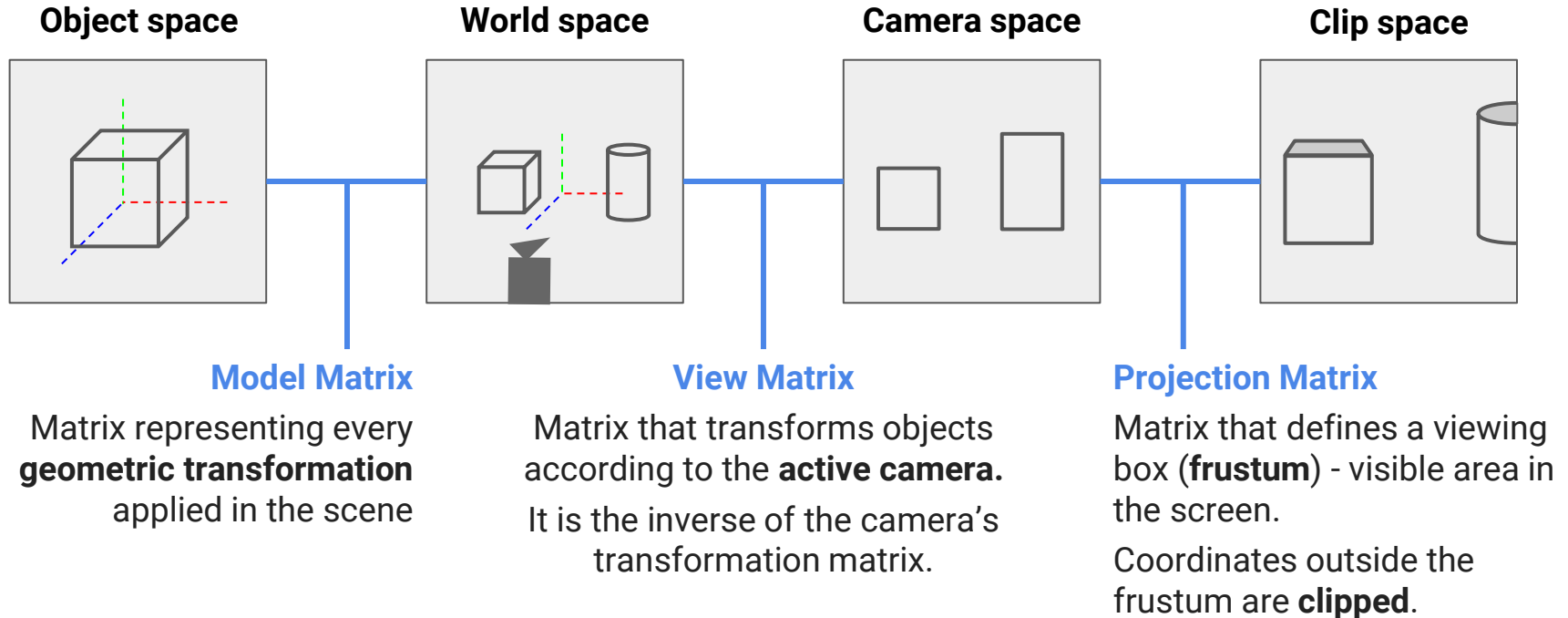
Applies transformations
related to camera frustum

Model-View Matrix

Represents transformations applied
to scene and camera position

**Input 3D position vector
for vertex** (as defined in
application)

Model-View-Projection Matrix



1 Creating Shaders - Fragment Shader

A fragment shader receives as **input**:

- Data from previous operations in the graphics pipeline (e.g., vertex shader)
- Custom data from application

A fragment shader creates as **output**:

- the **color for the current fragment**

1 Creating Shaders - Fragment Shader Example

```
struct lightProperties {  
    ...  
};
```

— Local struct for light properties
(position, ambient, diffuse,...)

```
#define N_LIGHTS 8
```

```
uniform lightProperties uLight[N_LIGHTS];
```

— Input array of lightProperties (length 8)

```
void main() {
```

— Main function, processes each fragment

```
    gl_FragColor = uLight[0].diffuse;
```

— Output fragment color

```
}
```


2 Loading Shaders using WebCGF library

The **WebCGF** library has a class for shaders - **CGFshader**

```
new CGFshader(gl, urlVertexShader, urlFragmentShader)
```


The scene has an **active shader** (default shader provided in library)

```
CGFscene.activeShader
```

Created shaders may be set as the scene's active shader

```
CGFscene.setActiveShader(CGFshader)
```

To set back to default, provide
`CGFscene.defaultShader`



3 Applying Shaders - Example

```
CGFscene.init(){  
    ...  
    this.shaderA = new CGFshader(...)  
}
```

Initializing shaders and other objects

```
...  
CGFscene.display(){  
    ...  
    this.setActiveShader(this.shaderA);
```

Scene setup (cameras, lights, matrices)

```
    ...  
    this.object.display();
```

Drawn elements affected by the custom shader

```
    ...  
    this.setActiveShader(this.defaultShader);
```

```
}
```

4 Passing Data from Application to Shaders

Data may be passed from the application to the *CGFshader* object

```
CGFshader.setUniformsValues(dictionary)
```

Key-value pair collection,
equivalent to JS object

This data is accessible in the shaders as uniform variables

```
uniform type variableKey;
```

The shaders may use this data to **transform the output**

4 Passing Scalar Data to Shader - Example

CGFscene

```
this.shaderA = new CGFshader(...)  
this.shaderA.setUniformsValues(  
    { scale: 2.0, key:value, ...});
```

— Initializing shaders and
passing uniforms' values

Vertex shader

```
uniform float scale;  
uniform valueType key;  
  
...
```

— Initializing the uniforms

```
void main() {  
    gl_Position = uPMatrix * uMVMMatrix * vec4(aVertexPosition*scale, 1.0);  
}
```

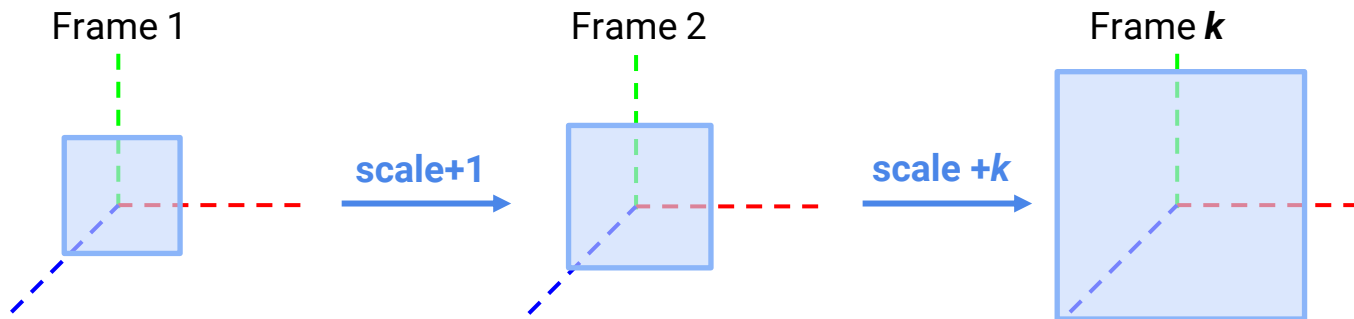
— Applying uniform data
to transform the output

4 Passing Scalar Data to Shader - Example

What happens if the value provided to the shader is altered periodically?

```
CGFscene.display(){  
  this.scale++;  
  this.shaderA.setUniformsValues({scale: this.scale});  
}
```

Sequence of frames with variation in object - Animation



4 Passing Texture to Shader - Example

CGFscene

```
display(){  
    ...  
    this.setActiveShader(this.shaderA);  
    this.texture0.bind();  
    this.object.display();  
}
```

Bind texture0 to WebGL context

Vertex/Fragment shader

```
uniform sampler2D uSampler;  
...
```

Where is uSampler defined?

And what if we want more than one texture?

4 Passing Multiple Textures to Shader

WebGL has a global array of references to textures – **texture units**

```
CGFtexture.bind(unit)
```

The ***unit*** parameter is the **texture unit** to which the texture is bound

By default, CGFscene passes the texture at `unit = 0` as `uSampler`

pseudo-code

```
activeShader.setUniformsValues({uSampler: 0});
```

4 Passing Multiple Textures - Example

```
CGFscene.init(){
    this.shaderA = new CGFshader(...)
    this.texture1 = new CGFtexture(...);
    this.texture2 = new CGFtexture(...);
    this.shaderA.setUniformsValues({texture2: 1});
}
CGFscene.display(){
    this.setActiveShader(this.shaderA);
    this.texture1.bind();
    this.texture2.bind(1);
    this.object.display();
    ...
}
```

— Passing as uniforms values the texture unit for the 2nd texture

— Binding textures to units 0 and 1

4 Passing Multiple Textures - Example

```
varying vec2 vTextureCoord;
```

texture coordinates from
vertex shader

```
uniform sampler2D uSampler;
```

Texture passed by CGFscene

```
uniform sampler2D texture2;
```

Texture passed by our code

```
void main() {  
    vec4 color = texture2D(uSampler, vTextureCoord);  
    ...  
    gl_FragColor = ...  
}
```

Shader function that retrieves
a texel from the sampler at
specified coordinates

Documentation and guides

Introduction to shaders using GLSL (presentation at Moodle)

GLSL Reference Card (available on Moodle)

WebCGF documentation for CGFshader

<https://paginas.fe.up.pt/~ruirodrig/pub/sw/webcgf/docs/#!/api/CGFshader>

Texture2D function

<https://thebookofshaders.com/glossary/?search=texture2D>

WebGL Shaders Tutorial

<https://webglfundamentals.org/webgl/lessons/webgl-shaders-and-glsl.html>