HOCHSCHULE FURTWANGEN UNIVERSITY | HFU

# CODE3

**A trip down the 3D graphics pipeline**

Summer semester 2026

Prof. Dr.-Ing. Uwe Hahne

HOCHSCHULE FURTWANGEN UNIVERSITY HFU

# Two triangles on their way through the graphics pipeline

(original slide set from Kristian Hildebrand, BHT Berlin)

# Learning goals

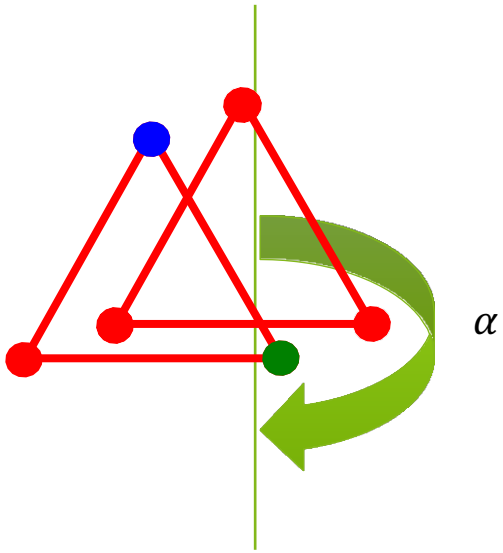
Understand in detail what happens during rendering.

The path of the data from the input into the computer to the display on the monitor.

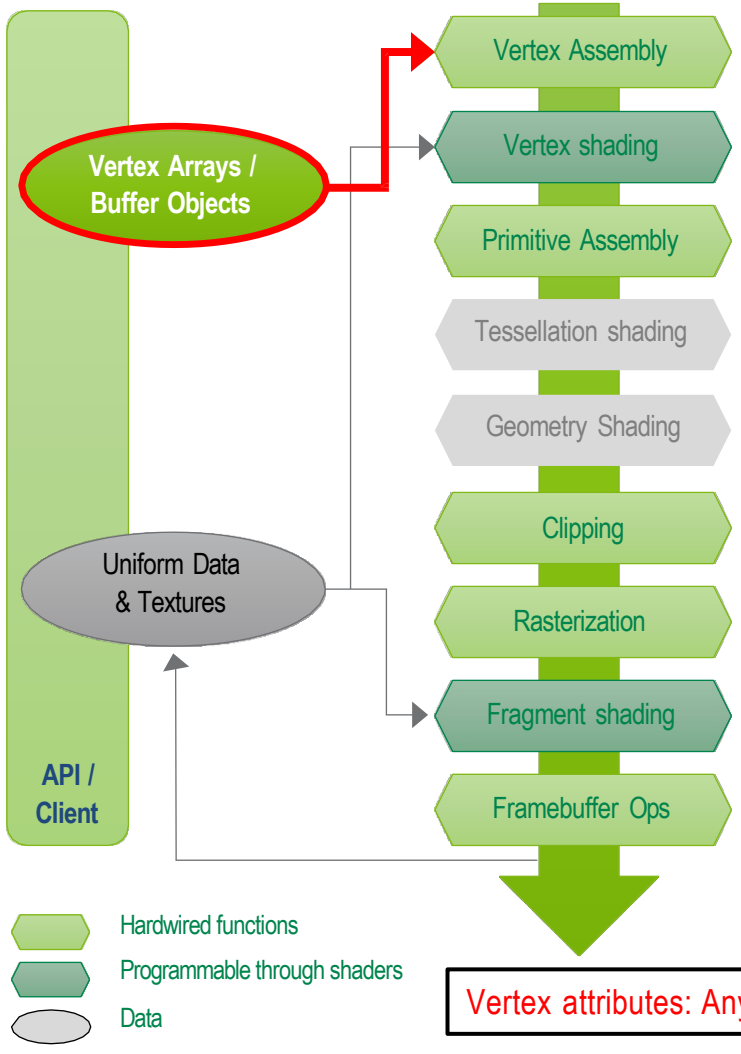
# Example application

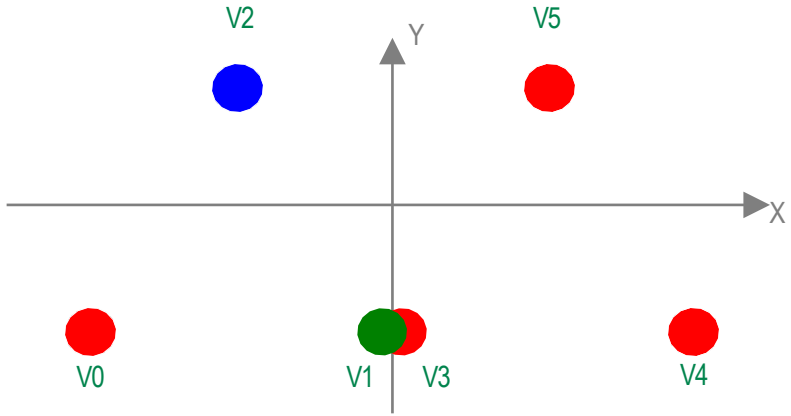
Given

- Scene: two triangles in 3D
- Positions of the triangle vertices
- Color per vertex (no lighting calculation etc.)
- Transformations: Rotation around the vertical axis by an angle



# Input data: Vertex attributes



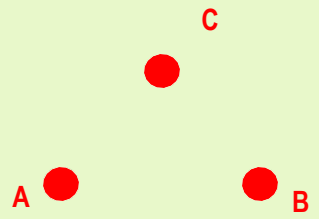

| ID | X | Y | Z | R | G | B |
|---|---|---|---|---|---|---|
| V0 | -1.0 | -0.5 | -2.0 | 1.0 | 0.0 | 0.0 |
| V1 | 0.0 | -0.5 | -2.0 | 0.0 | 1.0 | 0.0 |
| V2 | -0.5 | 0.5 | -2.0 | 0.0 | 0.0 | 1.0 |
| V3 | 0.0 | -0.5 | -3.0 | 1.0 | 0.0 | 0.0 |
| V4 | 1.0 | -0.5 | -3.0 | 1.0 | 0.0 | 0.0 |
| V5 | 0.5 | 0.5 | -3.0 | 1.0 | 0.0 | 0.0 |

Vertex attributes: Any data specified **per vertex**.

# Vertex Buffer Objects (VBO): *Attribute Buffer*

## Create attribute buffer VBO

```
const twoTrianglesVertices = [
          //  front triangle
          -1.0, -0.5, -2.0,
          0.0, -0.5, -2.0,
          -0.5, 0.5, -2.0,
          ...];
/*====== Define triangle buffer ======*/
const vertexBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(twoTrianglesVertices), gl.STATIC_DRAW);
```




And then again similarly for the color values.

## Activate VBO and bind to a shader variable

```
/* ==================== Connect the attribute with the vertex shader ============
// Locate the attribute from the vertex shader source in the program
const pointsAttributeLocation = gl.getAttribLocation(program, "vertex_points");
// Connect the attribute to the points data currently in the buffer object let size = 2;
let type = gl.FLOAT;
let normalized = false;
let stride = 0;
let offset = 0;
gl.vertexAttribPointer(pointsAttributeLocation, size, type, normalized, stride, offset);
// Send the points data to the GPU
gl.enableVertexAttribArray(pointsAttributeLocation);
```

Name of a variable in the Shader source code! GPU program

# Vertex attributes and gl.drawArrays()

What does gl.drawArrays() do with this VBO?

```
gl.drawArrays(gl.POINTS,   0,   3);
```

```
gl.drawArrays(gl.TRIANGLES, 0,  3);
```

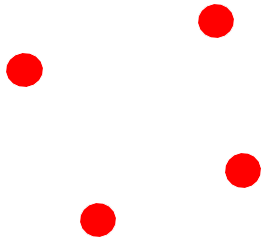
```
gl.drawArrays(gl.LINES,  0,  3);
```



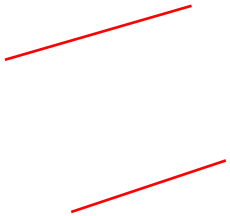
For each LINE **two vertices** needed!

To draw the outline of the triangle with lines, my VBO would have to contain would have to contain 3x2 vertices. This is not optimal.

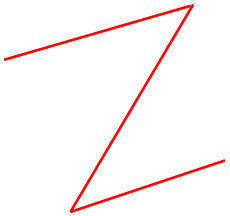
# Example: what can you do with four vertices?



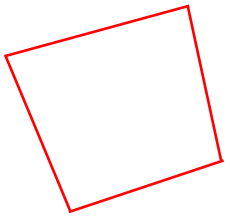
four points



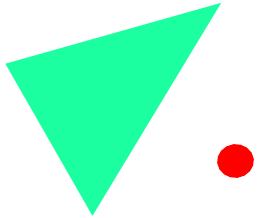
two lines



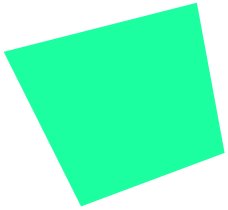
three lines



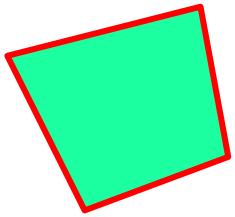
four lines



a triangle and a point



two triangles

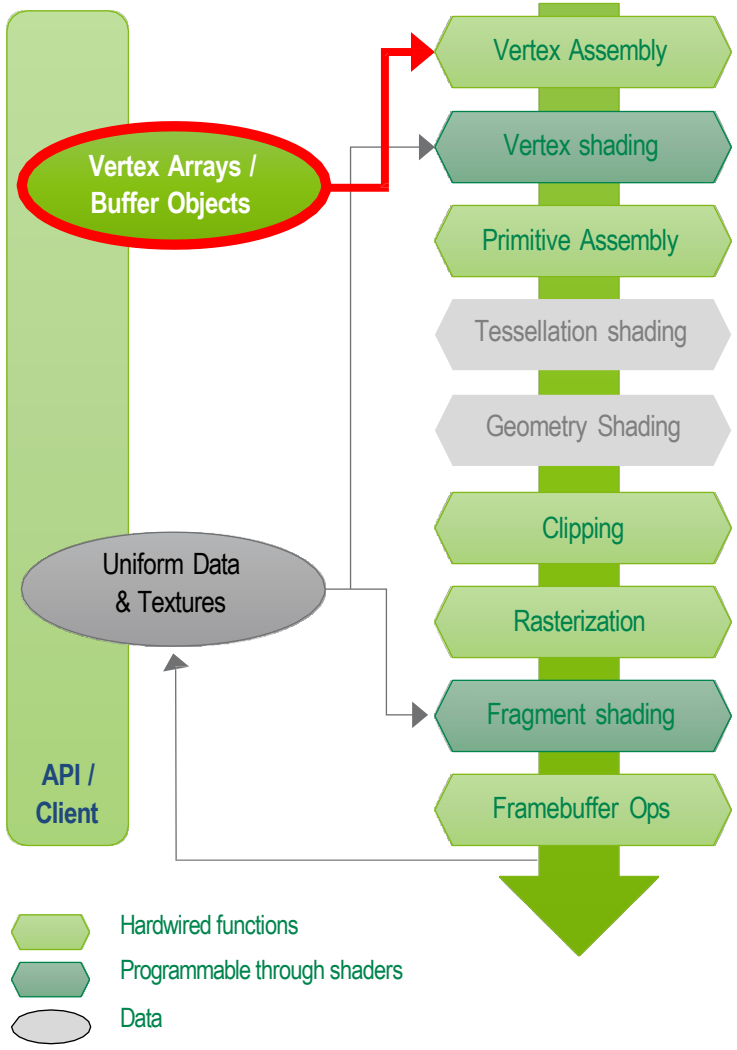


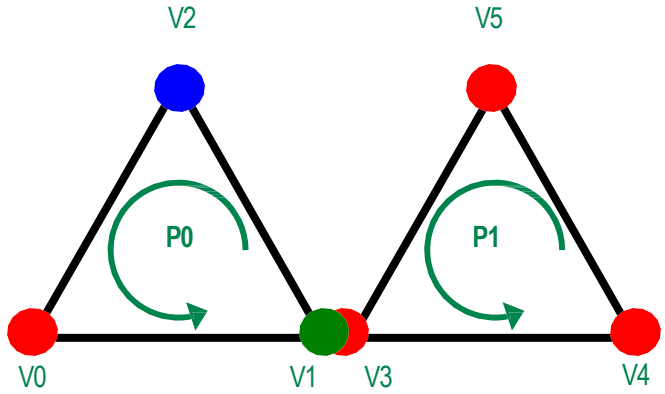
two triangles and four lines

. . .

The starting point is always the same four vertices!

# Input data: Vertex connectivity



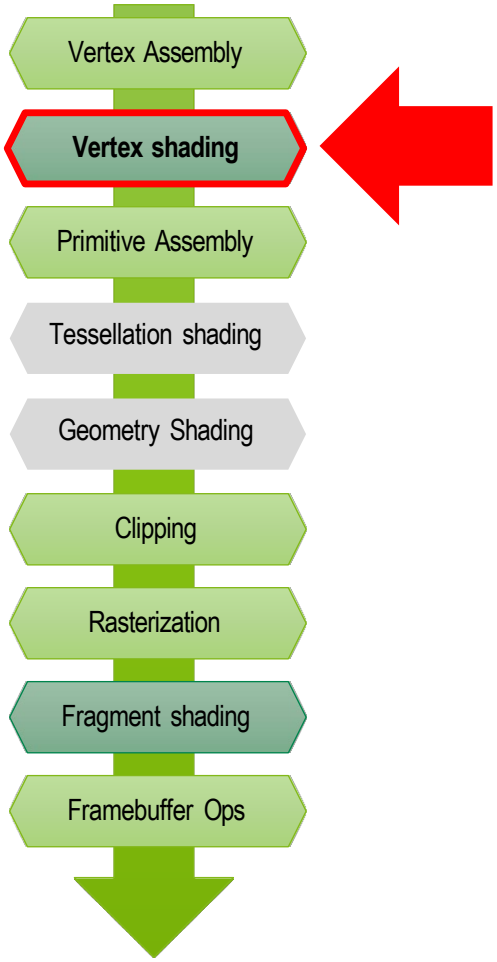




| Triangle | 1. vertex | 2. vertex | 3. vertex |
|---|---|---|---|
| P0 | 0 | 1 | 2 |
| P1 | 3 | 4 | 5 |

Vertex connectivity: Which vertices form a triangle?

Attention: the order of the vertices (winding order) defines where in a triangle is "front" and "back"!

# Configuration of the pipeline: Install vertex shader




```
#version 300 es

in vec4 aPosition;
in vec4 aColor;

uniform mat4 uModelViewMatrix;
uniform mat4 uProjectionMatrix;


out vec4 vFragColor;


void main() {
  gl_Position = uProjectionMatrix
        * uModelViewMatrix * aPosition;
  vFragColor = aColor;
}
```

Which attributes does a vertex have?

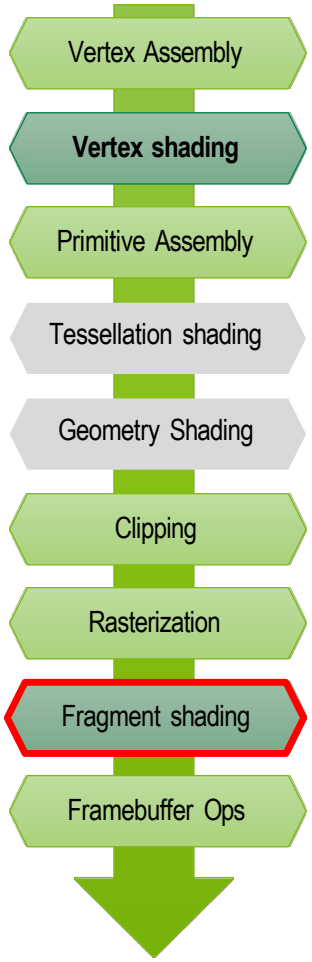
What global data does the shader expect?

What results are passed on?

Calculation rule

The application specifies what is required for each vertex should happen in the pipeline. This requires a vertex shader program can be specified.

# Configuration of the pipeline: Install fragment shader




**Attention:** If you use shaders in **Godot**, some additional variables and functions are already built-in by default. Check the documentation for details.

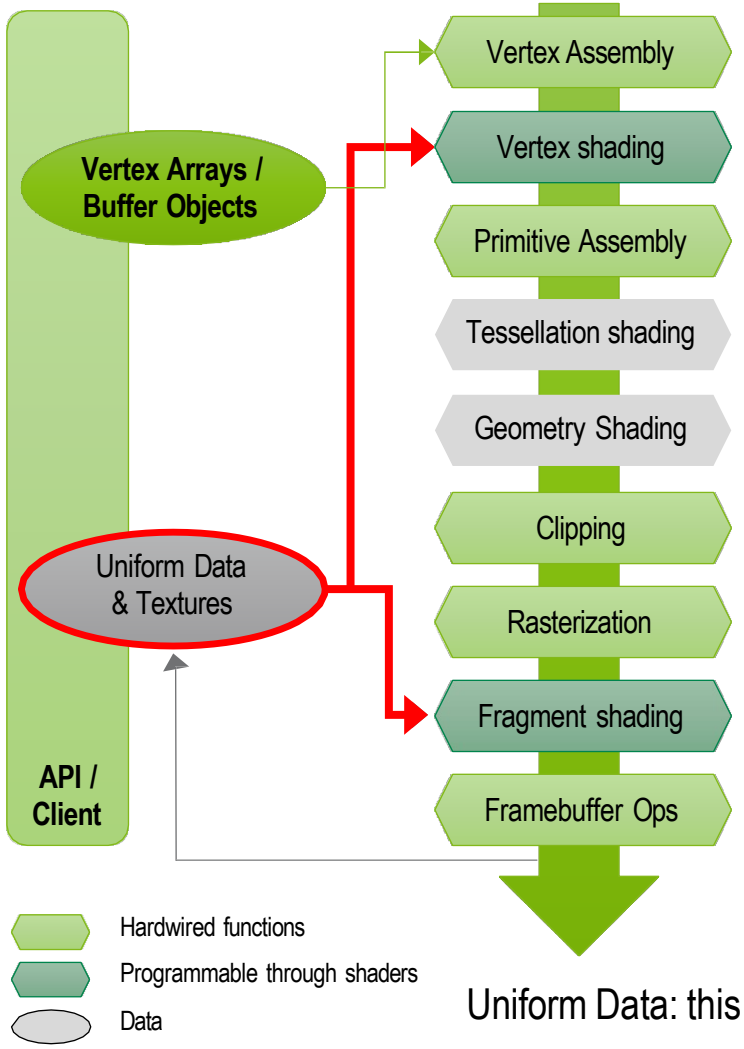
```
#version 300 es
precision mediump float;

in vec4 vFragColor;

out vec4 outColor; // pick any name

void main() {
  outColor = vFragColor;
}
```

Precision for WebGL

Results from the Vertex Shader

Calculation rule

# Uniform Data: Example transformations




| | | | |
|---|---|---|---|
| 0.98 | 0 | -0.2 | 0 |
| 0 | 1 | 0 | 0 |
| 0.2 | 0 | 0.98 | 0 |
| 0 | 0 | 0 | 1 |

`modelViewMatrix *`

Matrix determined Rotation of the scene and position of the camera

*** Values created with glMatrix.rotate(0.2,y)**

| | | | |
|---|---|---|---|
| 2.09 | 0 | 0 | 0 |
| 0 | 2.41 | 0 | 0 |
| 0 | 0 | -1 | -1 |
| 2.09 | 0 | -0.2 | 0 |

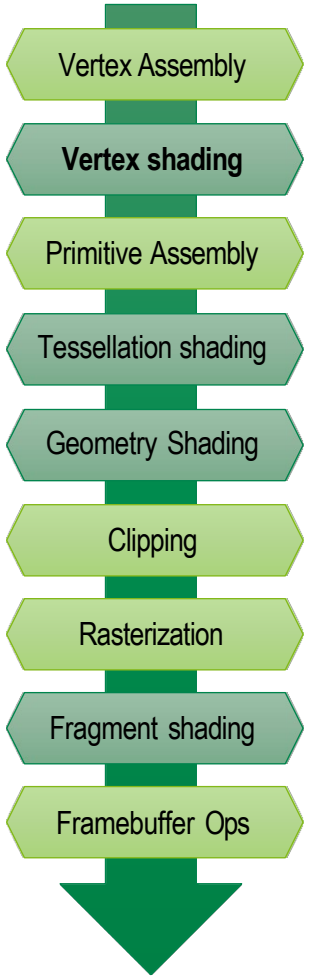
`projectionMatrix *`

Matrix determined Projection type (orthographic/ perspective)

*** Values created with glMatrix.perspective()**

Uniform Data: this data is **the same** for all vertices / fragments.

# Starting the pipeline functionality: with the drawing command



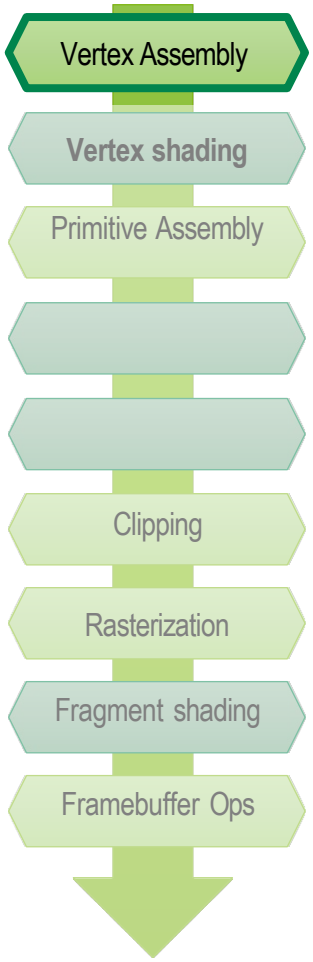

The pipeline is configured and the input data has been transmitted. But nothing is happening yet.

A **drawing command** must be sent by the application for it to actually start.

There are only two drawing commands:
**drawArrays()** and **drawElements()**

```
gl.clearColor(1, 1, 1, 1);
gl.enable(gl.DEPTH_TEST);
gl.depthFunc(gl.LEQUAL);
gl.clear(gl.COLOR_BUFFER_BIT |
gl.DEPTH_BUFFER_BIT);
// Draw the points on the screen
const mode = gl.TRIANGLES;
const first = 0;
const count = 6;
gl.drawArrays(mode, first, count);
```

# Vertex Assembly: Providing the vertex attributes




| ID | X | Y | Z | R | G | B | A |
|---|---|---|---|---|---|---|---|
| V0 | -1.0 | -0.5 | -2.0 | 1.0 | 0.0 | 0.0 | 1.0 |
| V1 | 0.0 | -0.5 | -2.0 | 0.0 | 1.0 | 0.0 | 1.0 |
| ... | | | | | | | |

Vertex 0

```
position=[-1.0,-0.5,-2.0]
color  = [1,0,0,1]
```

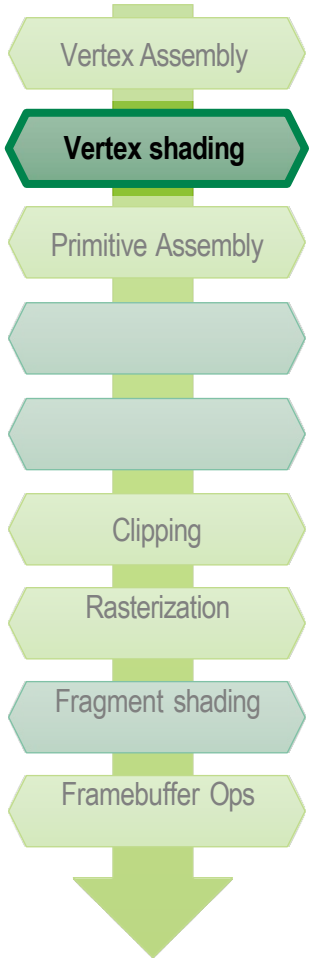
Vertex 1

```
position=[0.0,-0.5,-2.0]
color  = [0,1,0,1]
```

-- parallel for all vertices --

The vertex assembly compiles the associated data for each vertex. data for each vertex, which was transferred by the application using the vertex attribute arrays.

# Vertex shading: Execution of the shader program



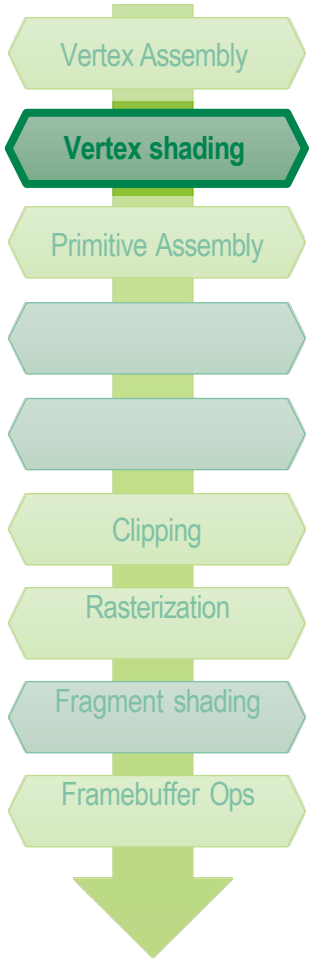

Vertex 0 -- parallel for all vertices -- Vertex 1

```
position=[-1.0,-0.5,-2.0]
color = [1,0,0,1]

gl_Position =
  uProjectionMatrix *
  uModelViewMatrix *
  aPosition;

vFragColor =
  aColor;
```

```
position=[0.0,-0.5,-2.0]
color = [0,1,0,1]

gl_Position =
  uProjectionMatrix *
  uModelViewMatrix *
  aPosition;

vFragColor =
  aColor;
```

This shader program multiplies `aPosition` first with the `uModelViewMatrix`, then with the `uProjectionMatrix`. It also passes `aColor` under the name `vFragColor` to the next shader in the pipeline (fragment shader)

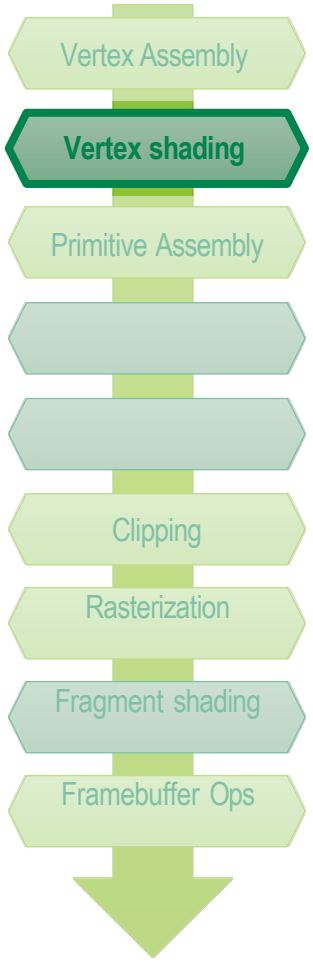
# Vertex Shading: Tasks



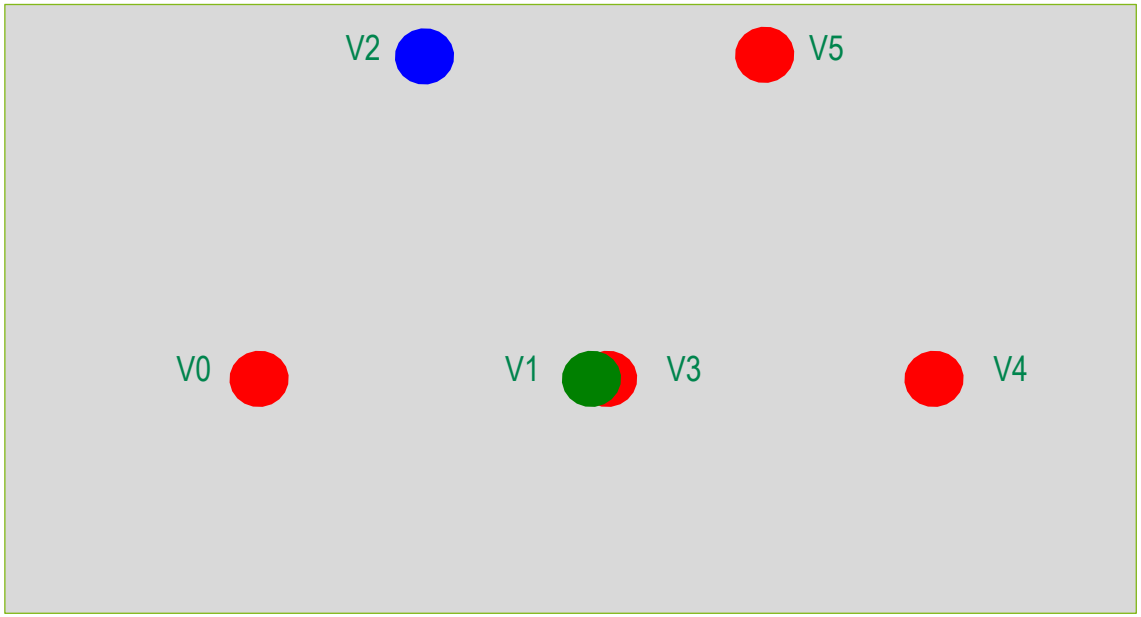

The main task of the vertex shader is to **transform** the position the position of each vertex into the so-called **clip coordinates**, in which the clipping later takes place.

The shader can transform any other data and **pass** it **on to the fragment shader**, such as the vertex color or texture coordinates.

# Result after vertex shading: Transformed vertices

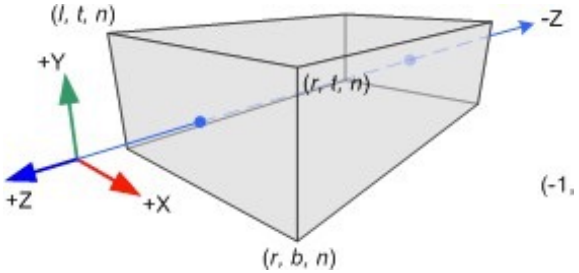



Orthographic projection

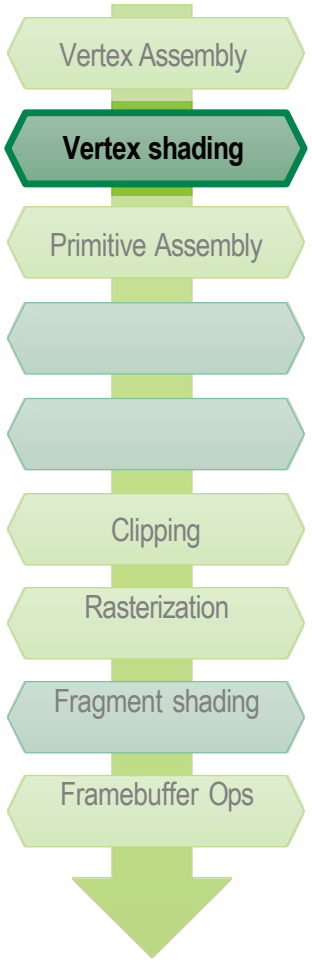



The projection has the task of mapping vertices from the 3D space onto the 2D image plane.

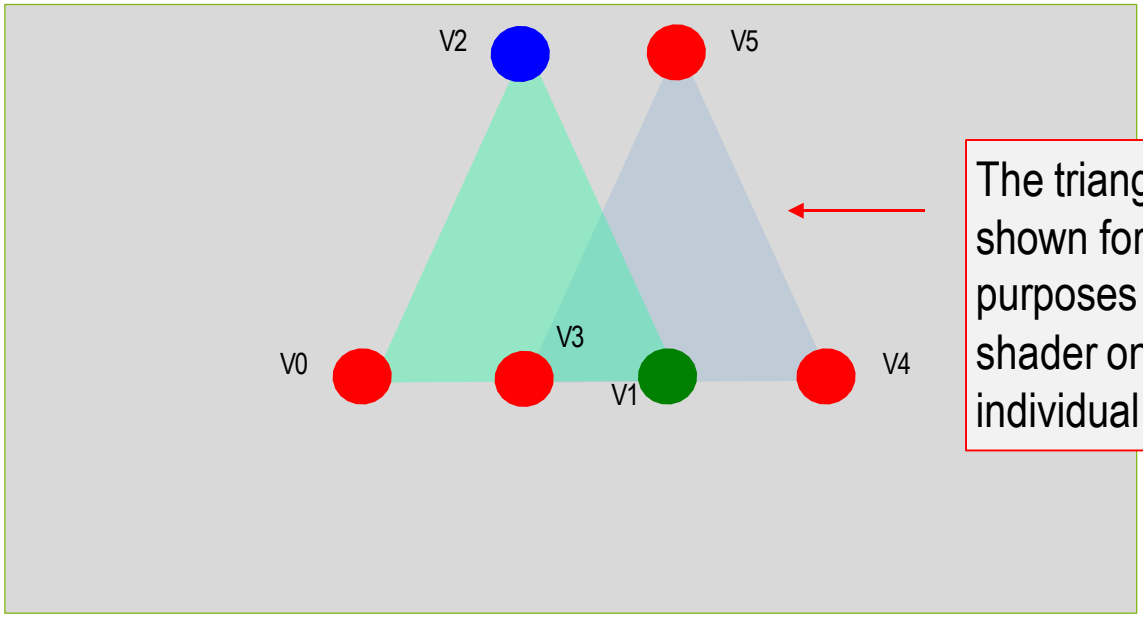
The orthographic projection does this, by ignoring the Z-coordinate.



# Result after vertex shading: Transformed vertices




Orthographic projection + rotation around Y-axis

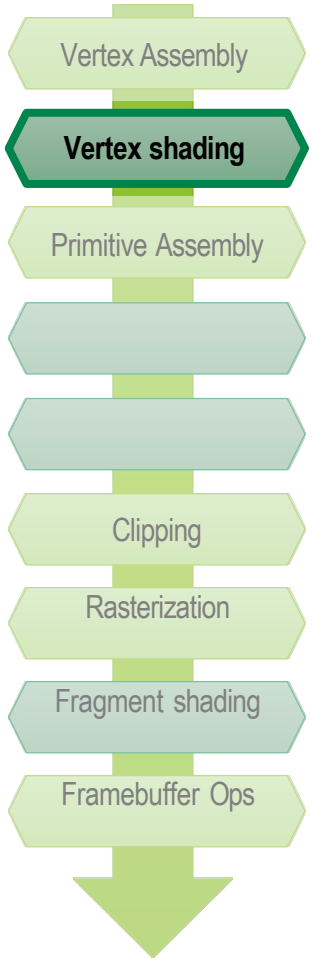



The triangles are only shown for illustration purposes only, the vertex shader only knows individual vertices!

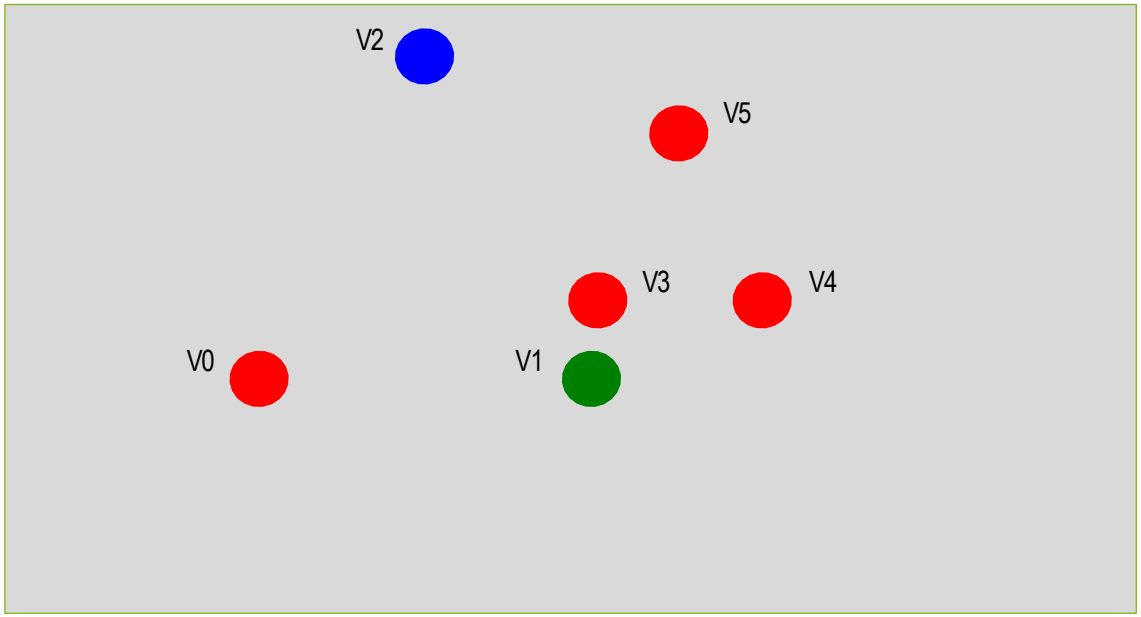
With the shader program used, the transformation depends depends **on the values of the modelView matrix**.

By changing the matrix, the application can display the scene rotated, for example, or move the camera through the scene.

# Result after vertex shading: Transformed vertices

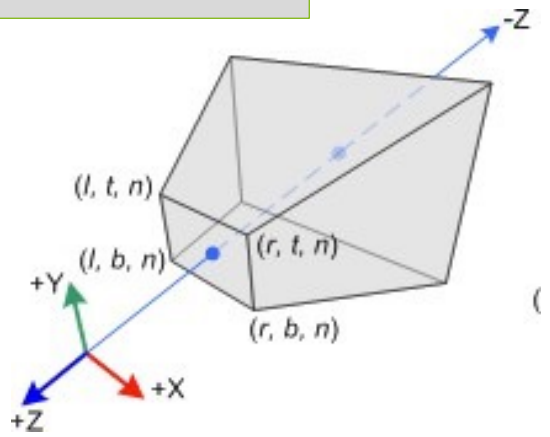



Perspective projection

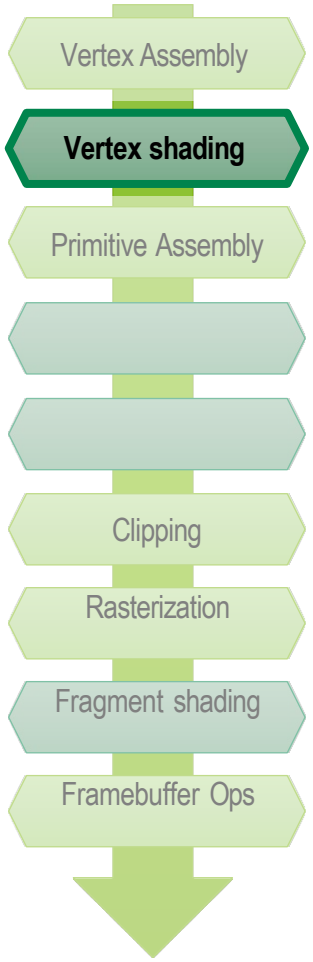



In perspective projection, the ray from the vertex to the eye is intersected with the image plane.

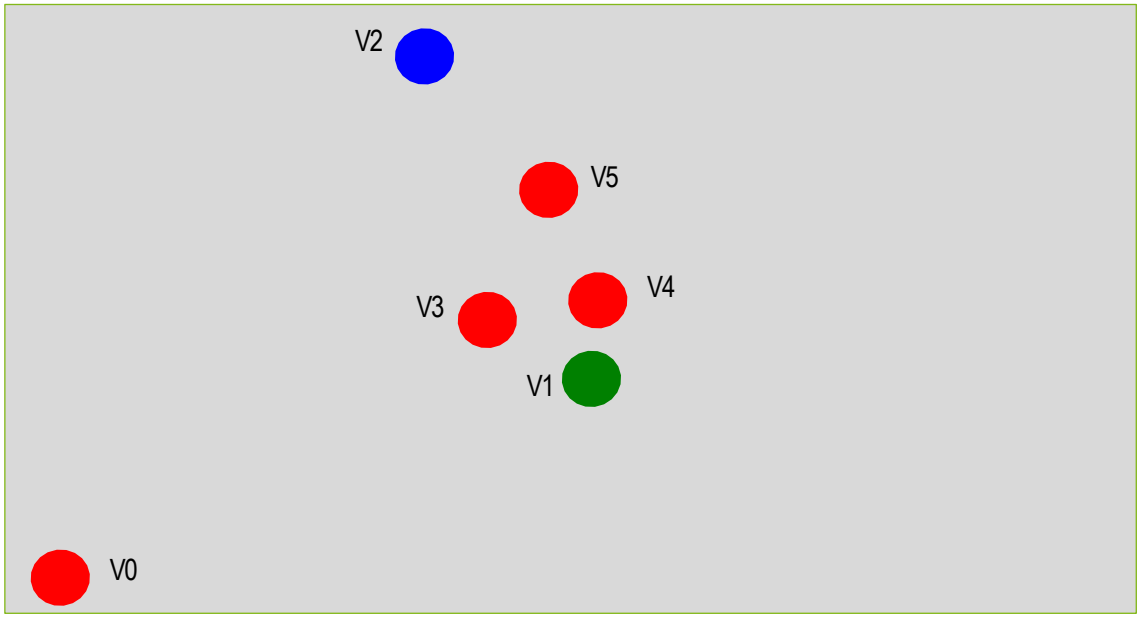
Attention, objects further back appear smaller appear smaller than those in front!



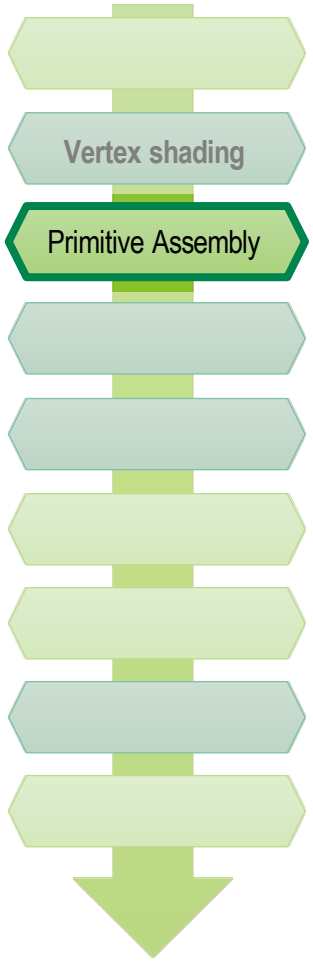
# Result after vertex shading: Transformed vertices

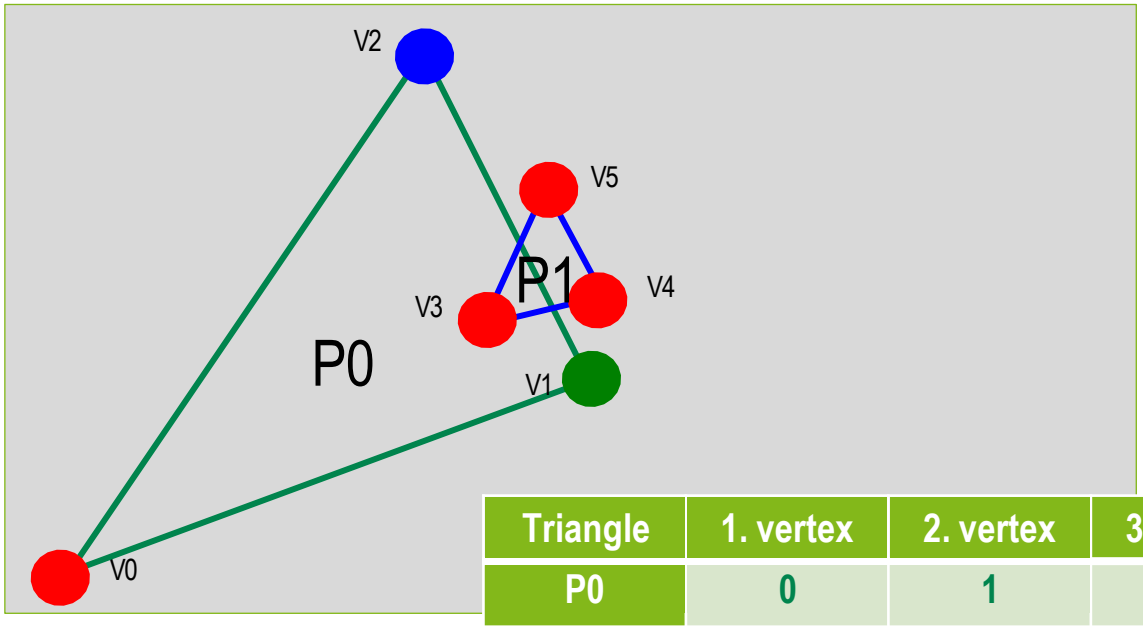



Perspective projection + 30°rotation



# Primitive assembly: Combining vertices into primitives

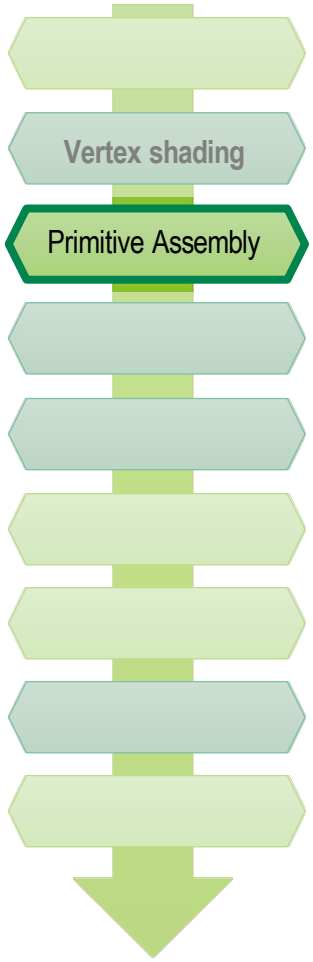


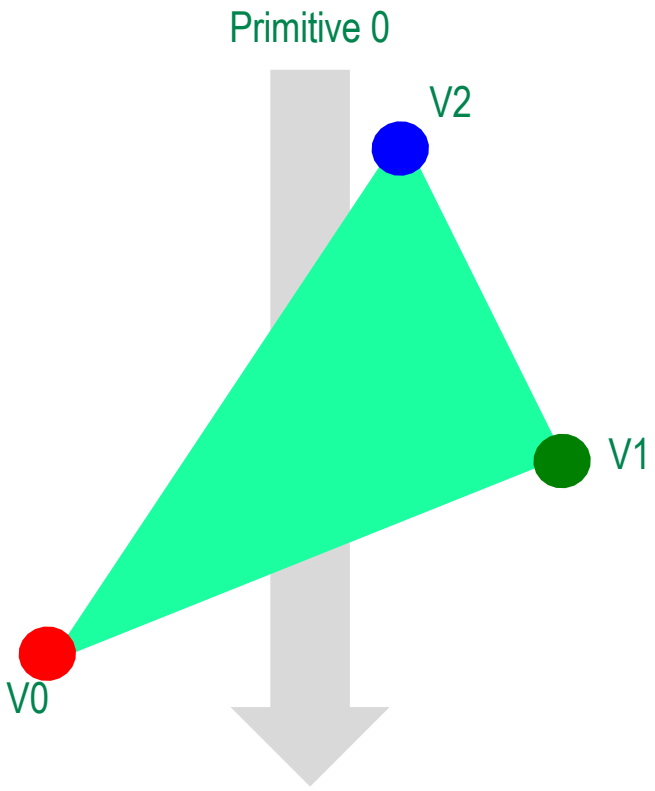

| Triangle | 1. vertex | 2. vertex | 3. vertex |
|---|---|---|---|
| P0 | 0 | 1 | 2 |
| P1 | 3 | 4 | 5 |

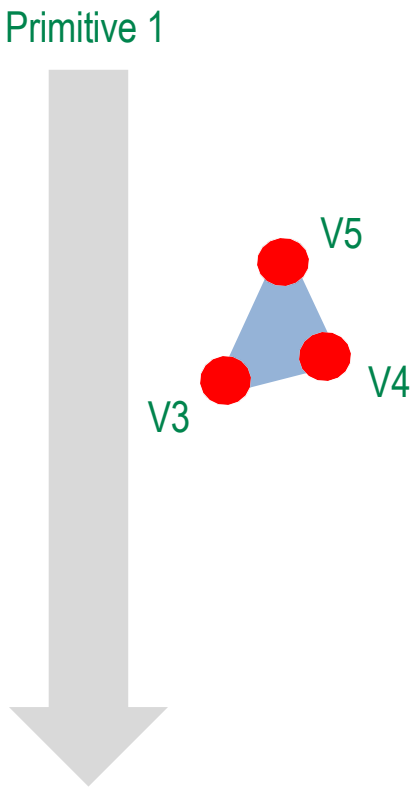
The *primitive assembly* searches for the corresponding vertices for each geometric primitive.

# Primitive assembly: Providing the vertices of a primitive

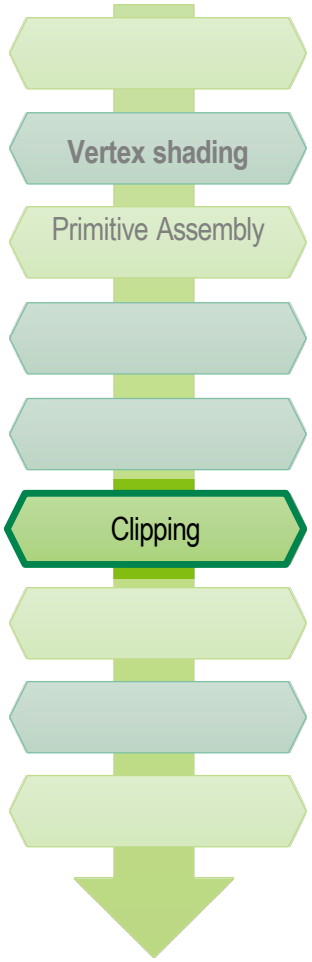


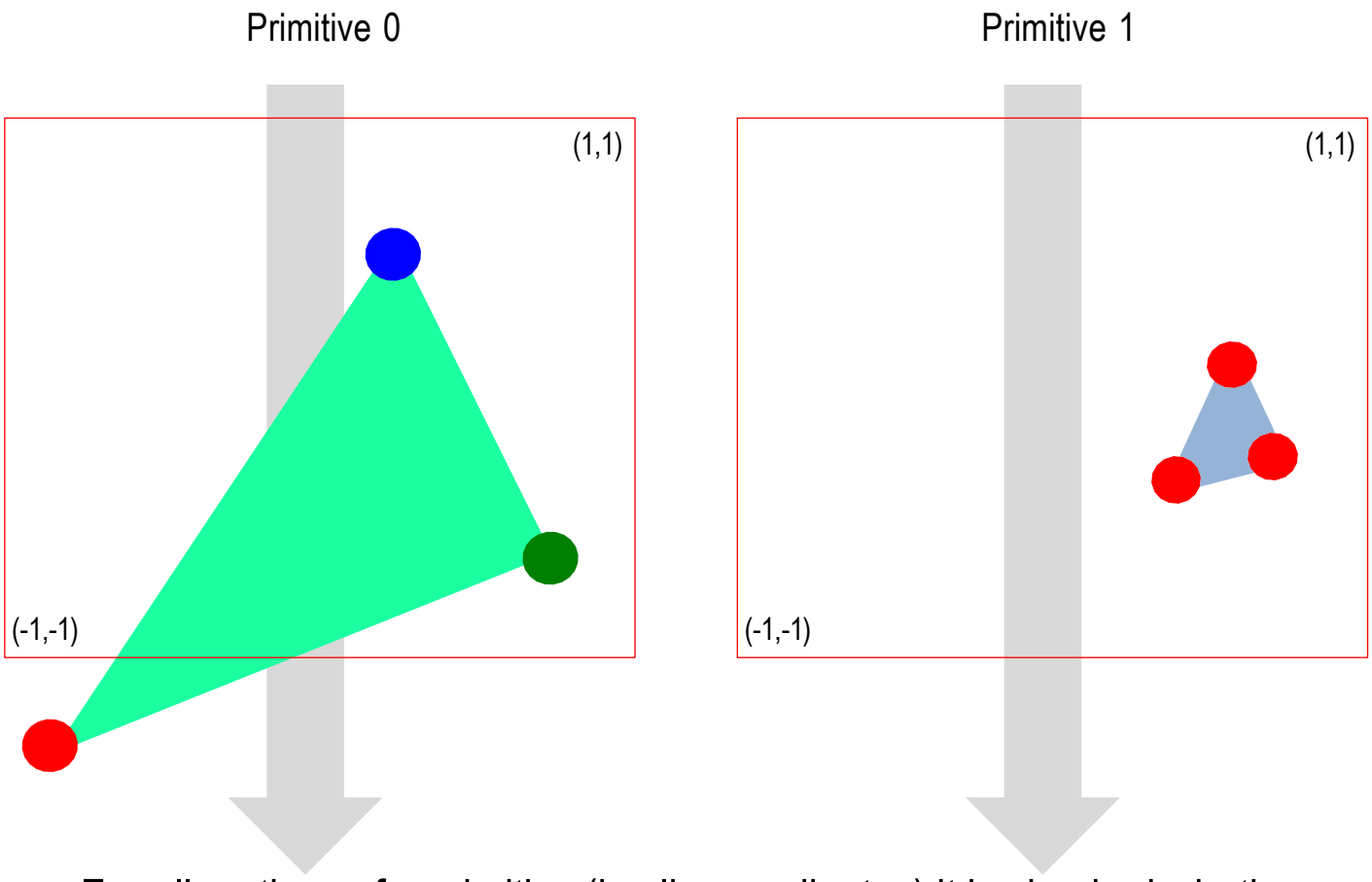

The next steps of the pipeline are no longer executed per vertex, but per primitive!



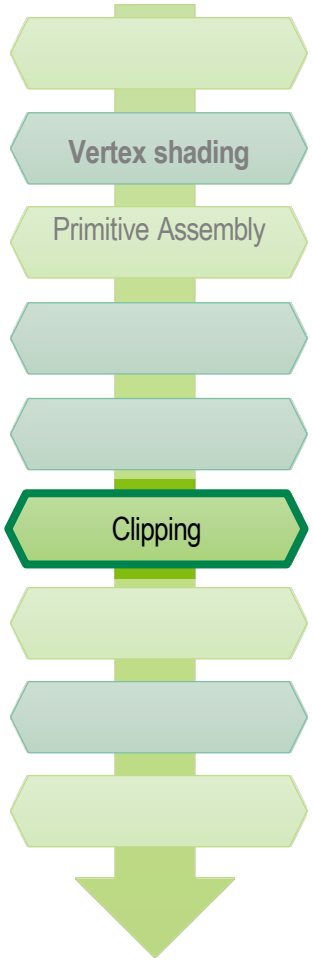
# Clipping: "Trimming" the primitives in the clipping window

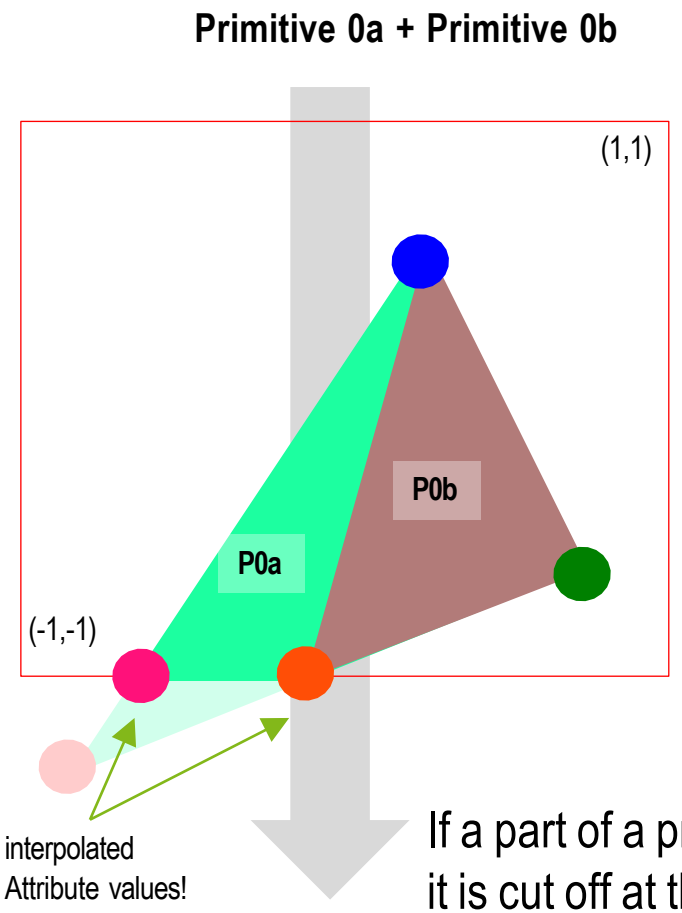


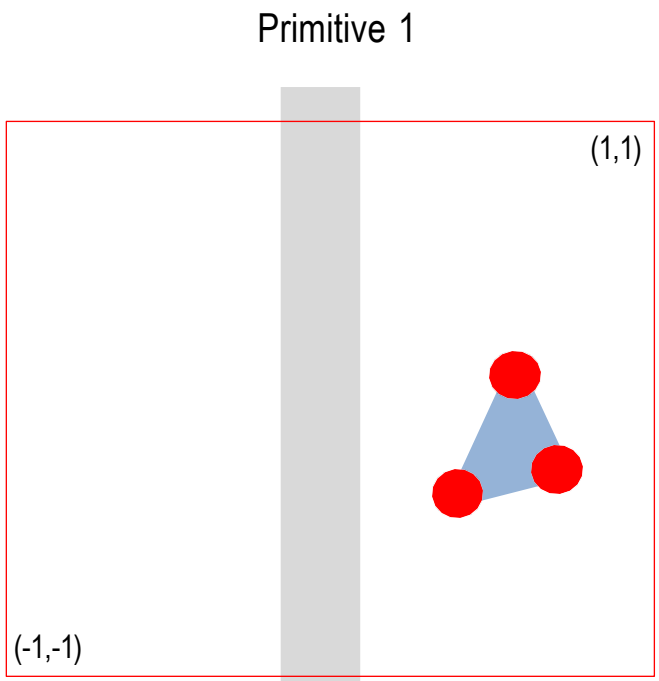




For all vertices of a primitive (in clip coordinates) it is checked whether the coordinate lies in the clip window [-1 : 1] in all dimensions (X, Y, Z).

# Clipping: "Trimming" the primitives in the clipping window

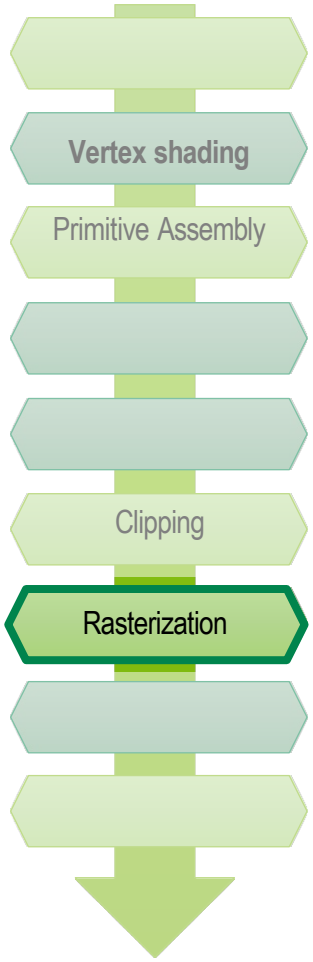


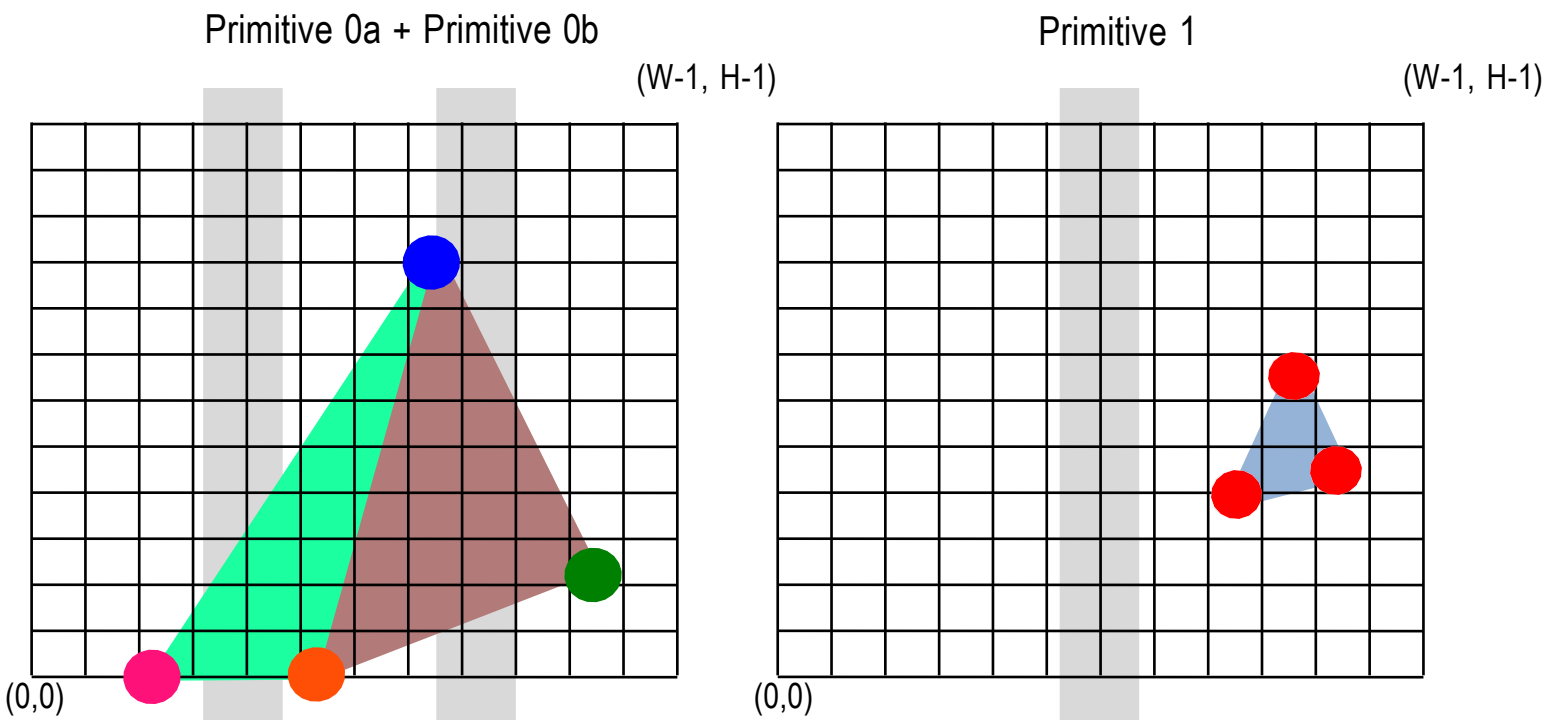




If a part of a primitive is outside the clip window, it is cut off at the edge of the window. This can create new vertices + primitives!

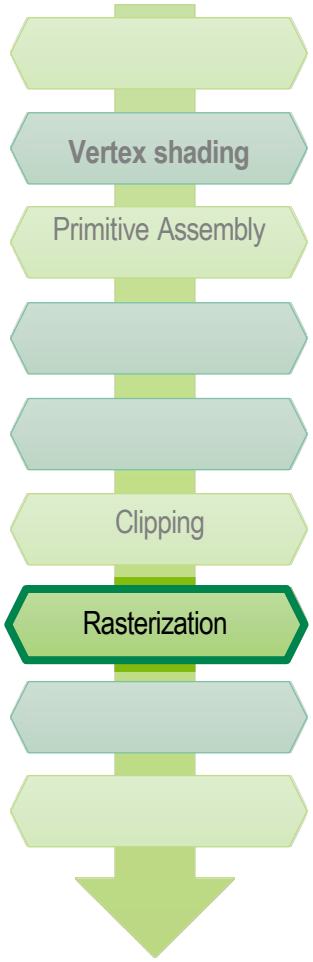
# Rasterization: Scanning of the primitives per pixel

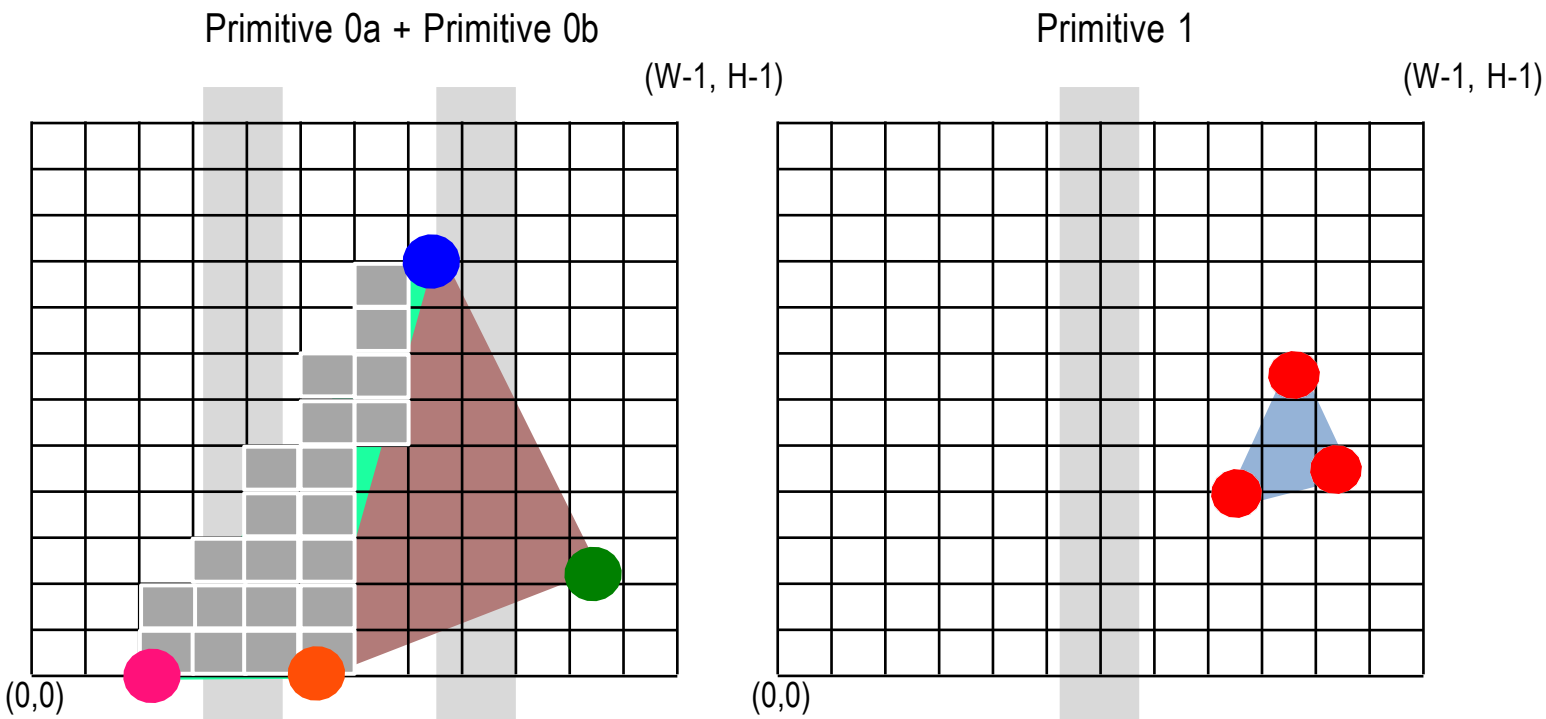



The pixel grid of the desired output window is applied to the primitive and for each pixel it is tested whether the primitive covers part of this pixel or not.

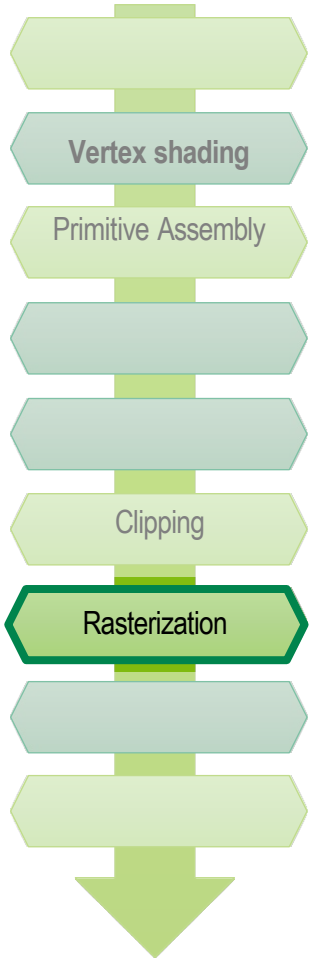
# Rasterization: Generation of fragments

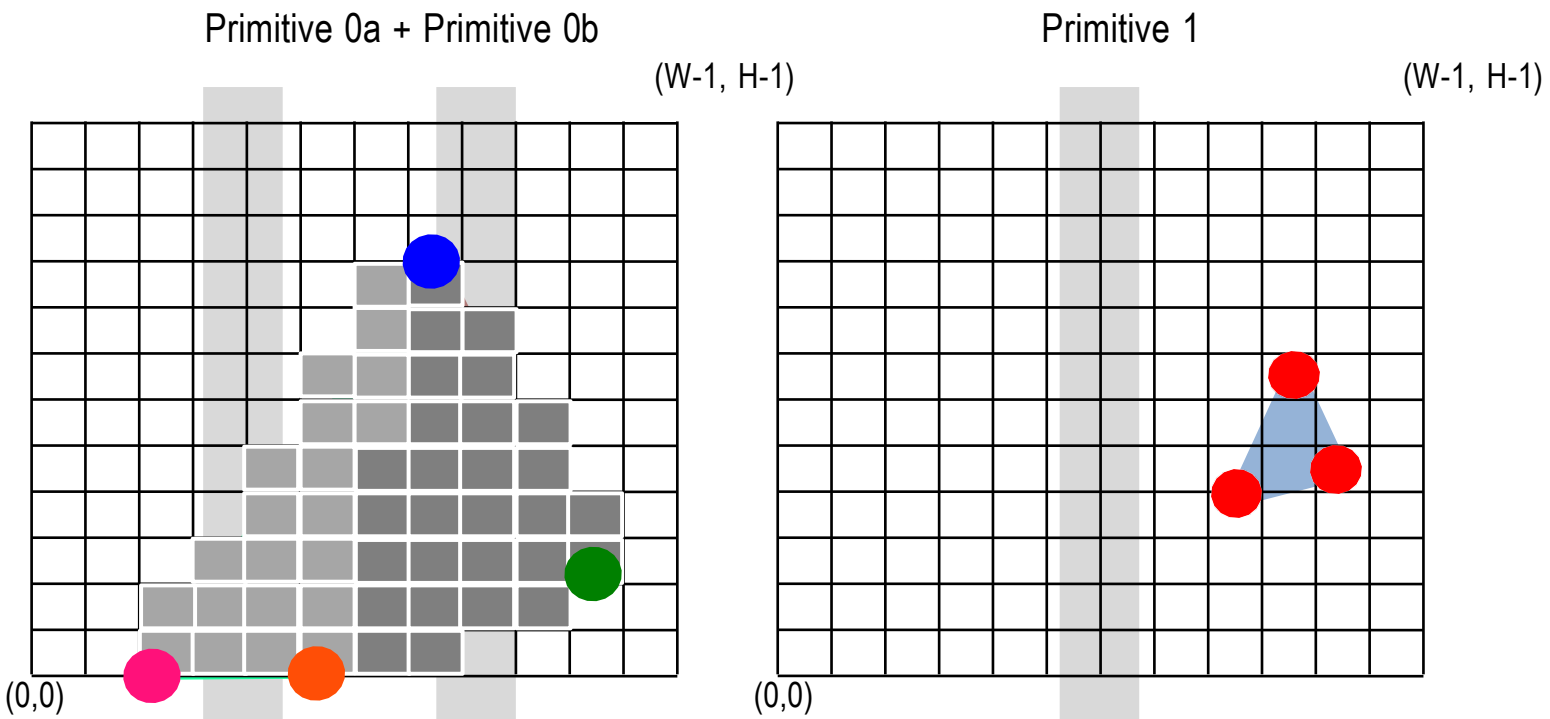



For each pixel that covers a part of the primitive, a **fragment** is generated. Each primitive generates its fragments independently of the others.

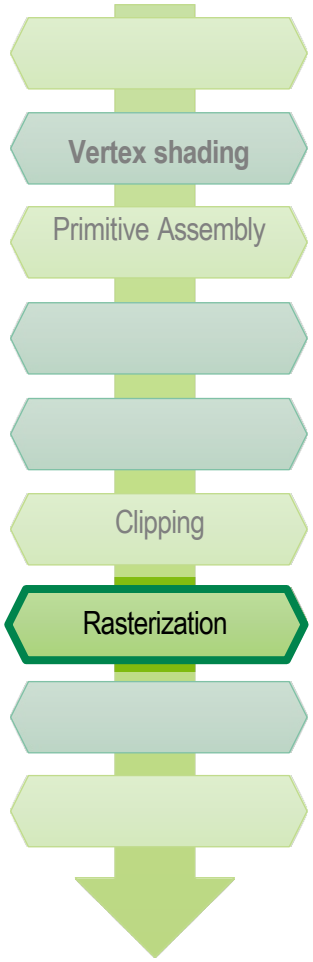
# Rasterization: Generation of fragments

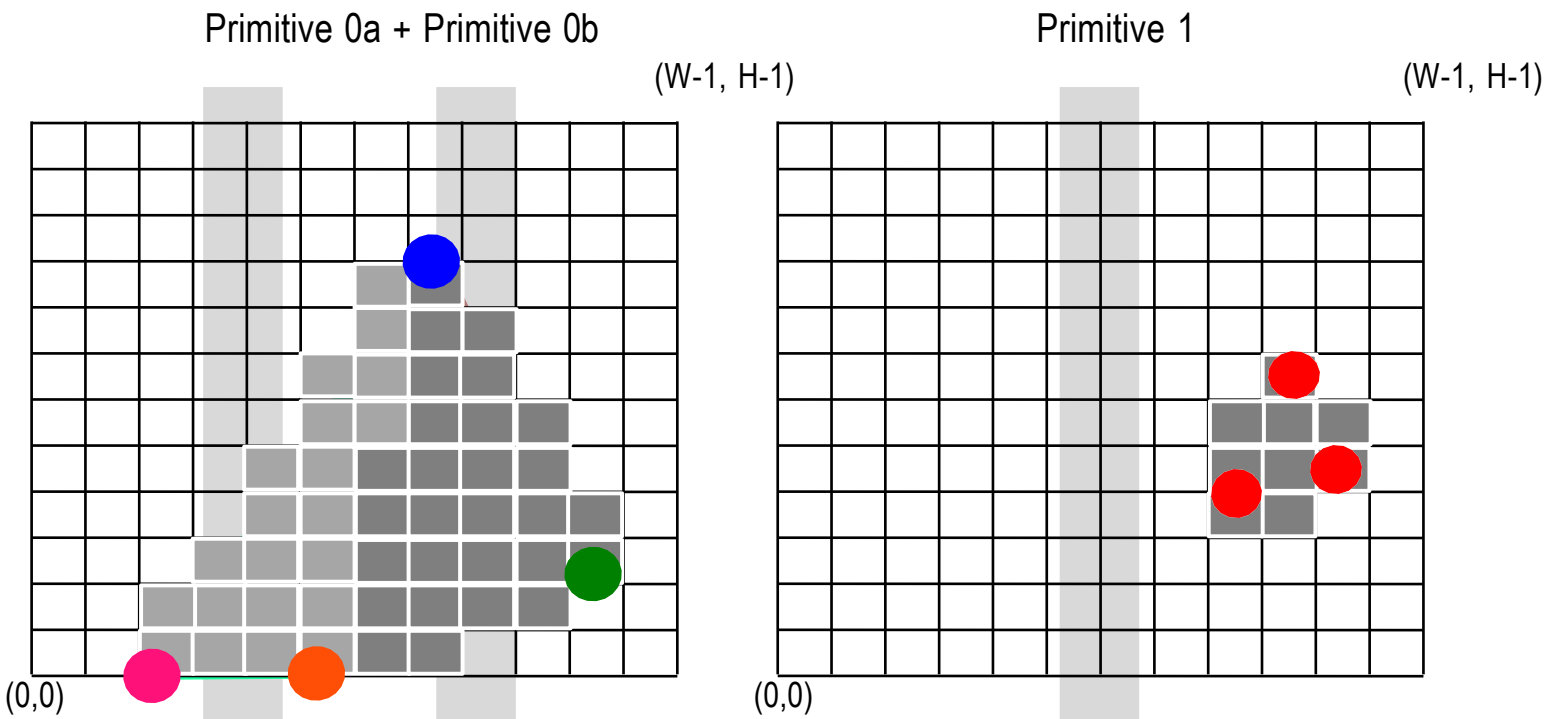



For each pixel that covers a part of the primitive, a **fragment** is generated. Each primitive generates its fragments independently of the others.

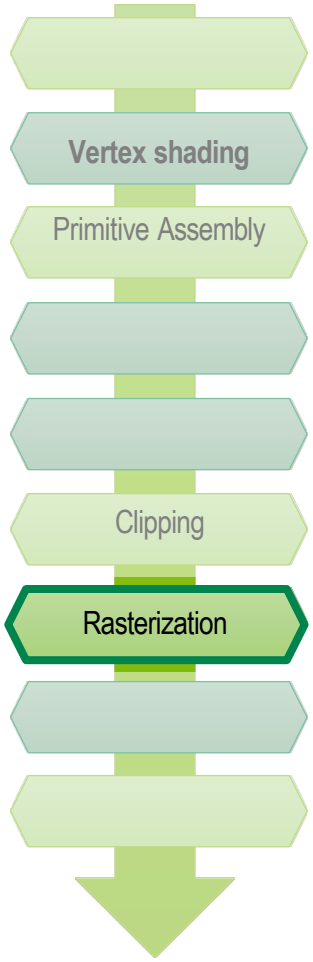
# Rasterization: Generation of fragments

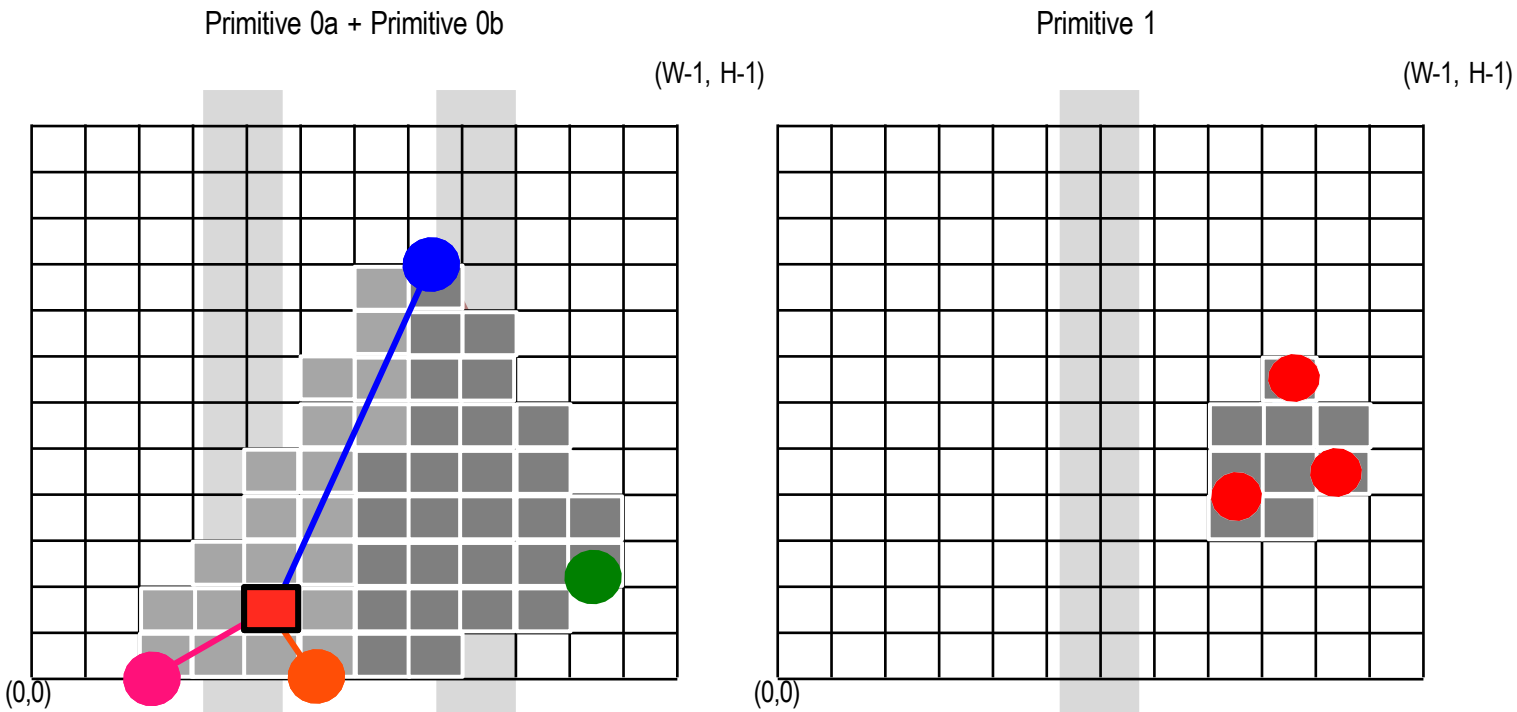



For each pixel that covers a part of the primitive, a **fragment** is generated. Each primitive generates its fragments independently of the others.

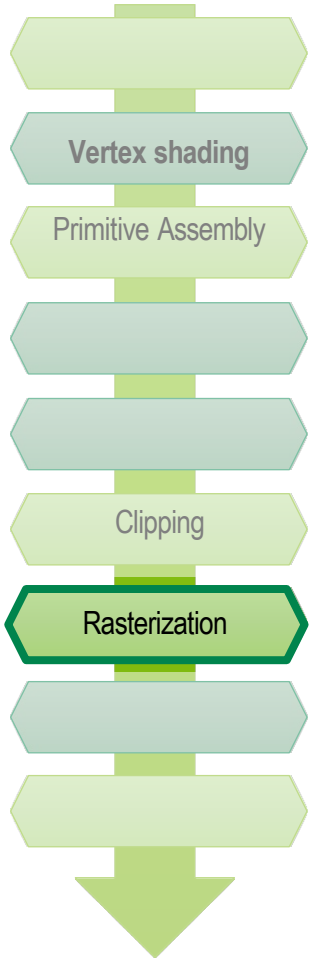
# Rasterization: Interpolation of vertex attributes

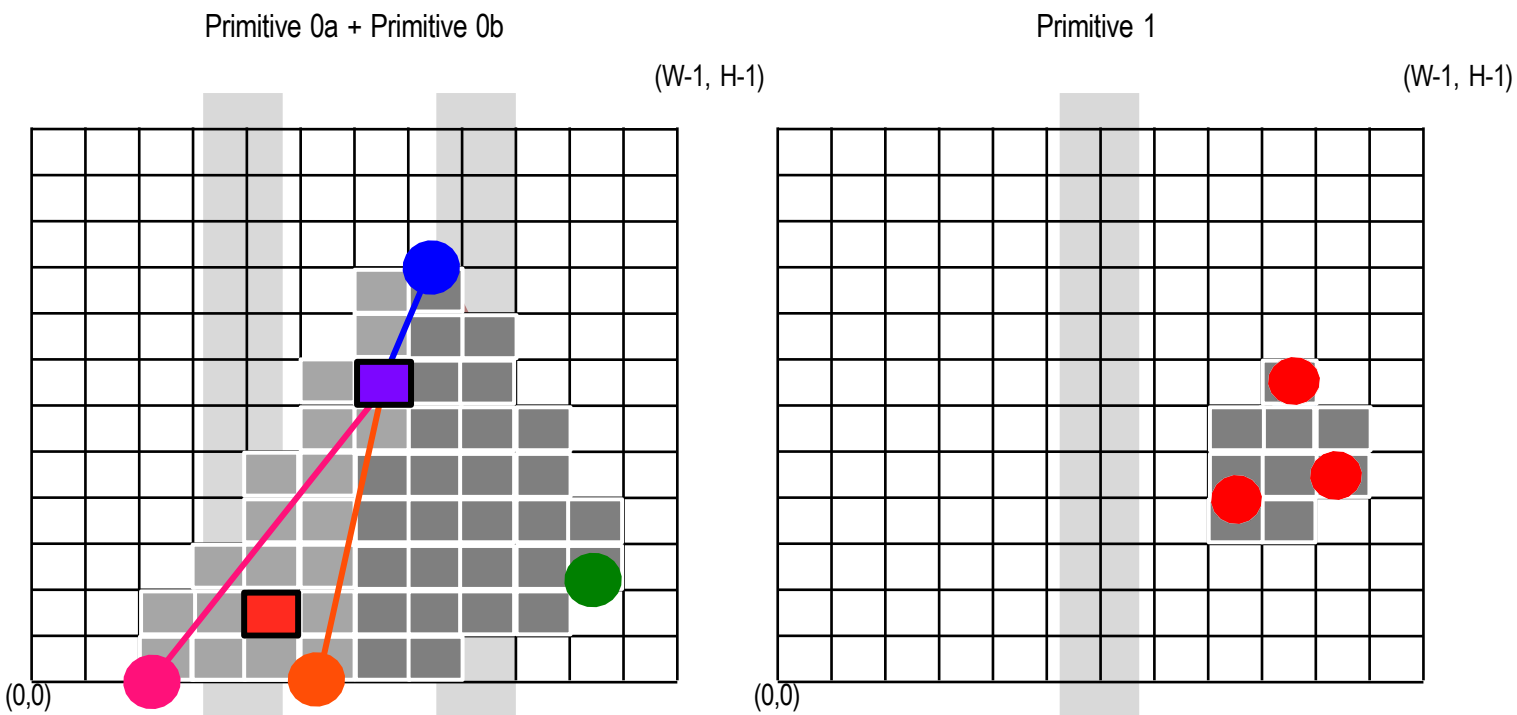



The attributes of the fragment required for the shader program are then calculated. This is done by **barycentric interpolation** of the corresponding vertex attributes of the primitive.

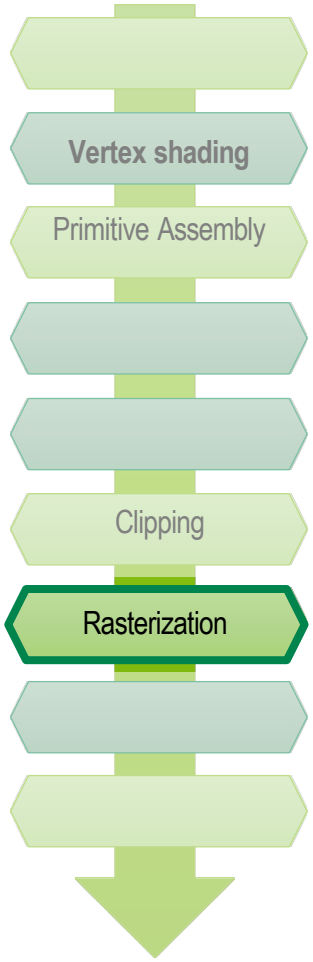
# Rasterization: Interpolation of vertex attributes

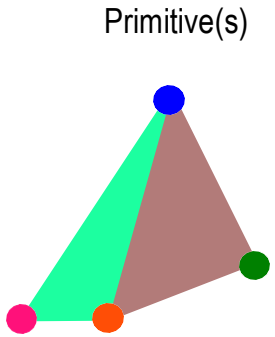


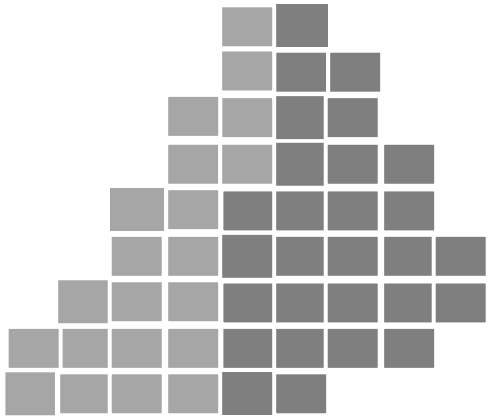

The attributes of the fragment required for the shader program are then calculated. This is done by barycentric interpolation of the corresponding vertex attributes of the primitive.

# Result of rasterization and interpolation: fragments




Rasterization



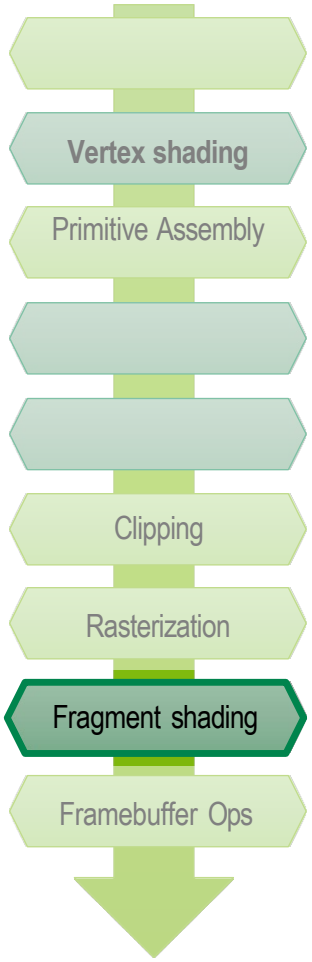


-- from now on the fragments are processed in parallel --

```
struct  fragment  {
  x,y:  int;  //  position  in  window  coords
  r,g,b,a:  float;  //  color/alpha  values
  depth:  float;  //  Z  buffer  value  from  0.0  to  1.0
  texX,  texY:  float;  //  texture  coordinates
  nX,  nY,  nZ:  float;  //  normal  direction  in  eye  coords
  ...
}
```

A fragment can contain many attributes!

# Fragment shading: Execution of the shader program




Fragment 0

```
#version 300 es
precision mediump float;


in vec4 vFragColor;
out vec4 outColor;

void main() {
  outColor = vFragColor;
}
```

Fragment 1

```
#version 300 es
precision mediump float;


in vec4 vFragColor;
out vec4 outColor;

void main() {
  outColor = vFragColor;
}
```

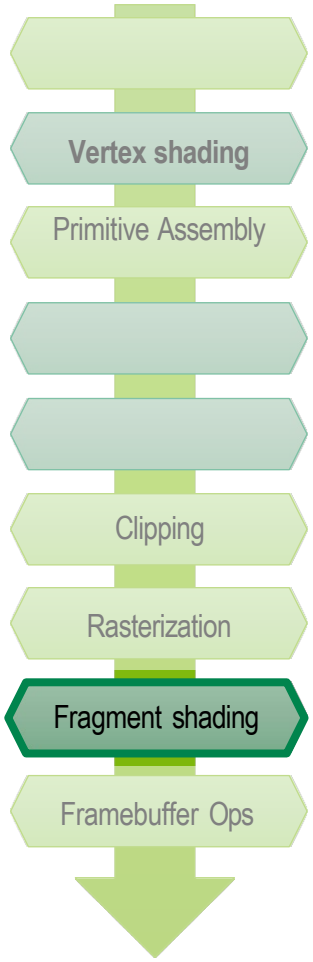
Fragment 2 ...

```
#version 300 es
precision mediump float;


in vec4 vFragColor;
out vec4 outColor;

void main() {
  outColor = vFragColor;
}
```

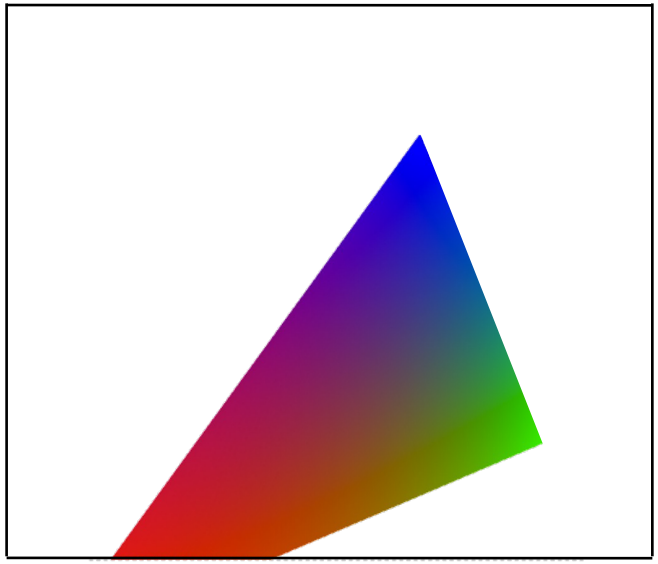
This very simple shader program assigns the value of the `vFragColor` attribute to each fragment as the final color

-- parallel for all fragments --

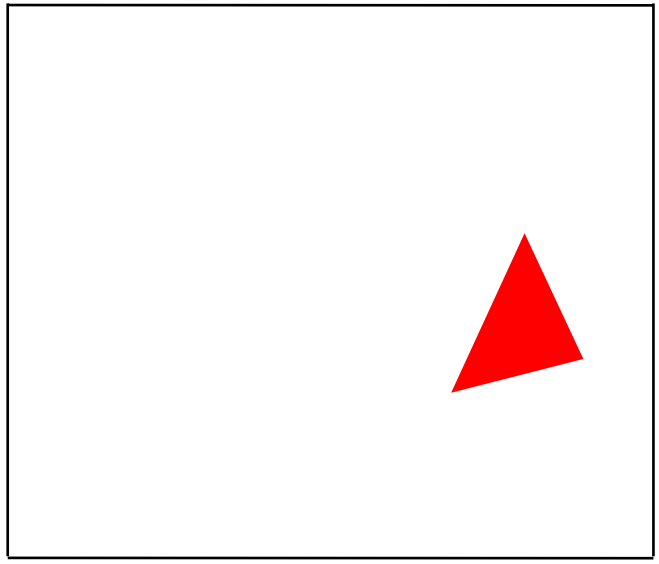
# Fragment Shading: Result




Fragments generated by primitive 0

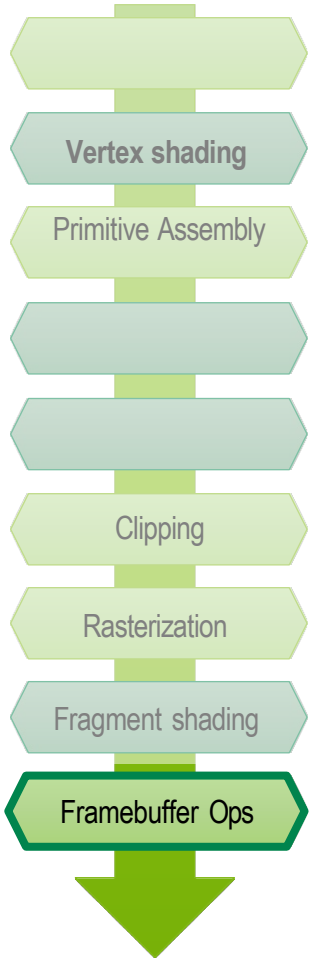


Fragments generated by primitive 1

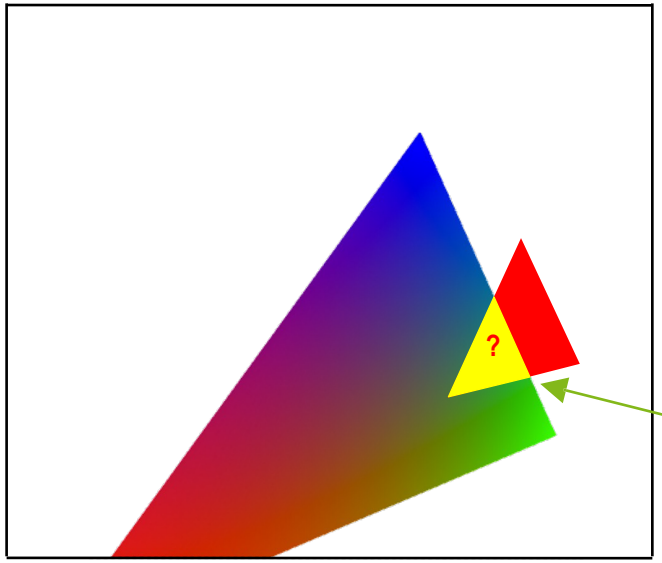


The barycentric interpolation of the vertex colors creates linear color gradients within the triangle.

# Framebuffer operations (a.k.a. *compositing*, *blend & merge*)



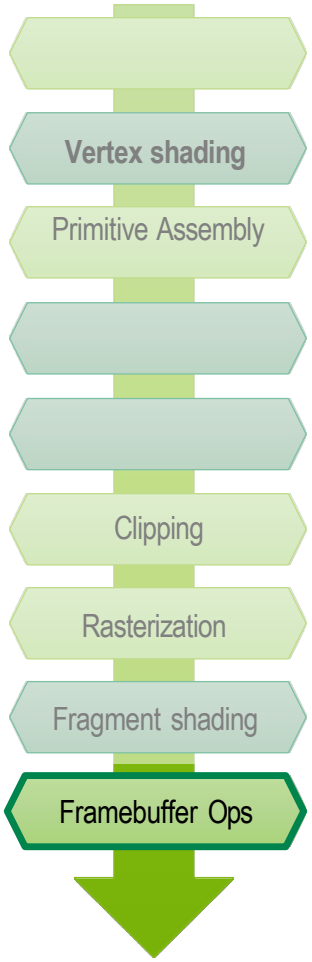

Fragments of two primitives



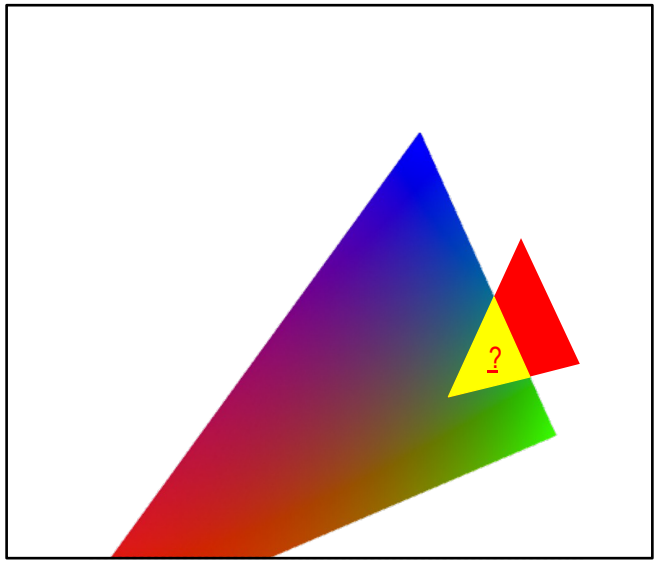

Here the fragments of two Primitive on the same pixels!

The framebuffer operations assign each fragment to its pixel and decide what influence the fragment has on the final color of the pixel. This is particularly important if different fragments fall on the same pixel!

# Framebuffer operations: Z-buffer




Fragments of two primitives




## Z-buffer

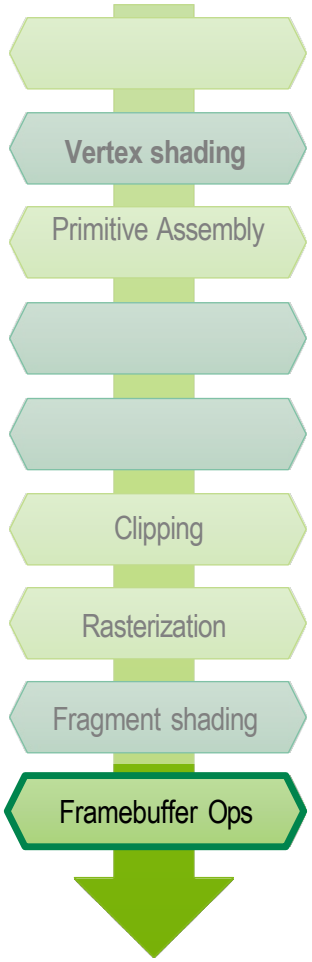
For all fragments of primitive 0:

- Write the color of the fragment to the color buffer.
- Write the Z value ("depth") of the fragment to the Z buffer.

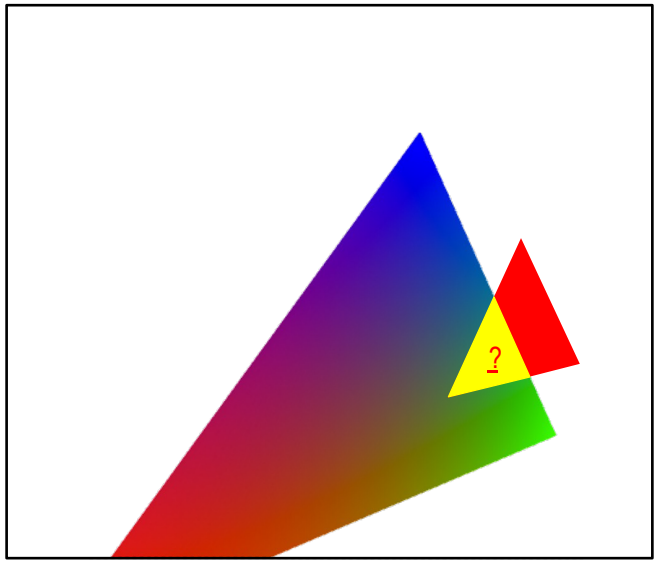
For each fragment of primitive 1:

- Compare the Z value of the fragment with that of the corresponding pixel.
- If fragment is behind previous pixel, ignore it.

# Framebuffer operations: Alpha Blending




Fragments of two primitives




## Alpha Blending

If primitive semi-transparent

- Color value of the current fragment must be mixed with the previous pixel color
- Sequence (which primitive is drawn first in the frame buffer) is important

# Summary: Graphics pipeline

**Vertex Buffer Objects (VBO)**
Geometry and material data

| ID | X | Y | Z | R | G | B | A |
|---|---|---|---|---|---|---|---|
| V0 | -1.0 | -0.5 | -2.0 | 1.0 | 0.0 | 0.0 | 1.0 |
| V1 | 0.0 | -0.5 | -2.0 | 0.0 | 1.0 | 0.0 | 1.0 |
| V2 | -0.5 | 0.5 | -2.0 | 0.0 | 0.0 | 1.0 | 1.0 |
| V3 | 0.0 | -0.5 | -3.0 | 1.0 | 0.0 | 0.0 | 1.0 |
| V4 | 1.0 | -0.5 | -3.0 | 1.0 | 0.0 | 0.0 | 1.0 |
| V5 | 0.5 | 0.5 | -3.0 | 1.0 | 0.0 | 0.0 | 1.0 |

| Dreieck | 1. Vertex | 2. Vertex | 3. Vertex |
|---|---|---|---|
| P0 | 0 | 1 | 2 |
| P1 | 3 | 4 | 5 |

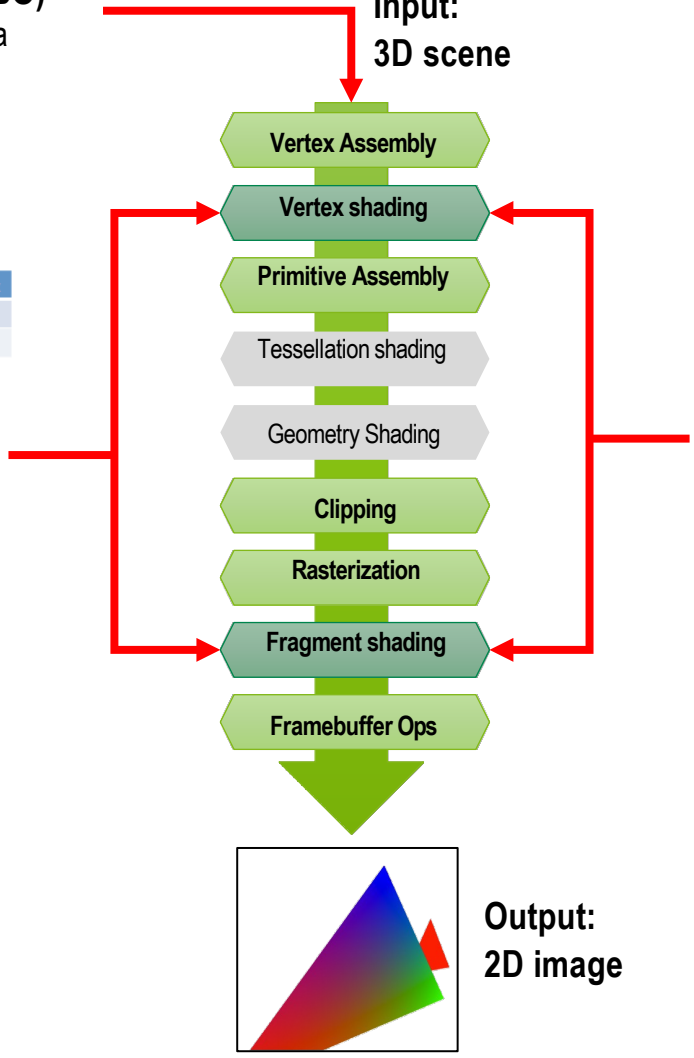
**Uniform Data**
global constants

| | | | |
|---|---|---|---|
| 0.98 | 0 | -0.2 | 0 |
| 0 | 1 | 0 | 0 |
| 0.2 | 0 | 0.98 | 0 |
| 0 | 0 | 0 | 1 |

`modelViewMatrix`

| | | | |
|---|---|---|---|
| 2.09 | 0 | 0 | 0 |
| 0 | 2.41 | 0 | 0 |
| 0 | 0 | -1 | -1 |
| 2.09 | 0 | -0.2 | 0 |

`projectionMatrix`




Vertex Shader

```
attribute vec4 aPosition;
attribute vec4 aColor;
uniform mat4 uModelViewMatrix;
uniform mat4 uProjectionMatrix;
varying vec4 vFragColor;


void main() {
  gl_Position = uProjectionMatrix *
          uModelViewMatrix *
          aPosition;
  vFragColor = aColor;
}
```

**GPU programs**
compiled shader code

```
precision mediump float;

varying vec4 vFragColor;

void main() {
  gl_FragColor = vFragColor;
}
```

Fragment Shader