ARLAB

ME/CprE/ComS 557

Computer Graphics and Geometric Modeling

Introduction to GPU Programming

September 1st, 2015

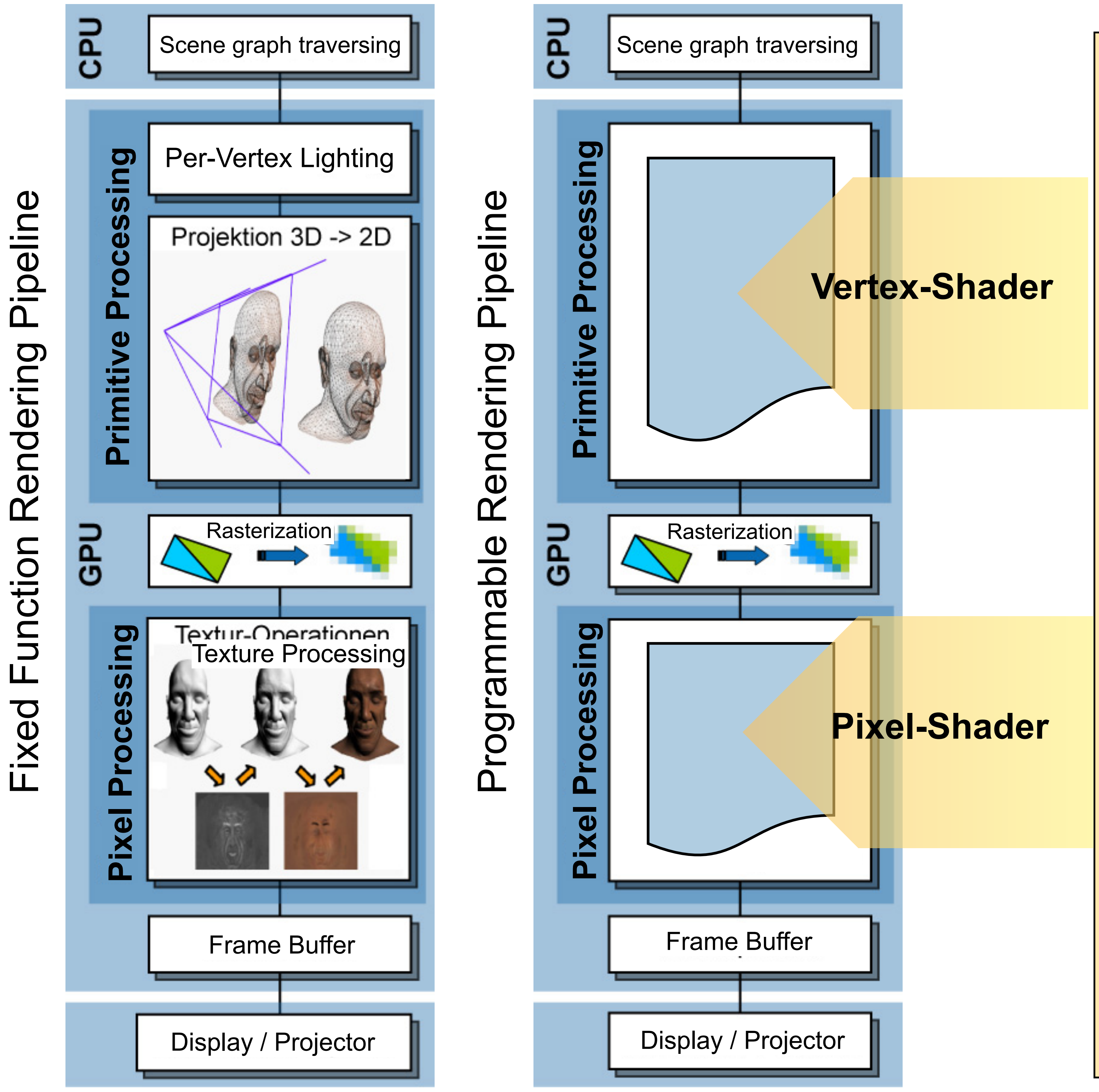
Rafael Radkowski

VRAC|HCI IOWA STATE UNIVERSITY OF SCIENCE AND TECHNOLOGY

Topics

- Concepts of GPU-Programming
- Vertex and Pixel (Fragment) Shader
- Functions and Built-in variables

Concept of GPUs



The **fixed function rendering pipeline** implements all necessary calculations (transformation & lighting, rasterization, shading, etc.) via hardware circuits to generate 3D graphic on display.

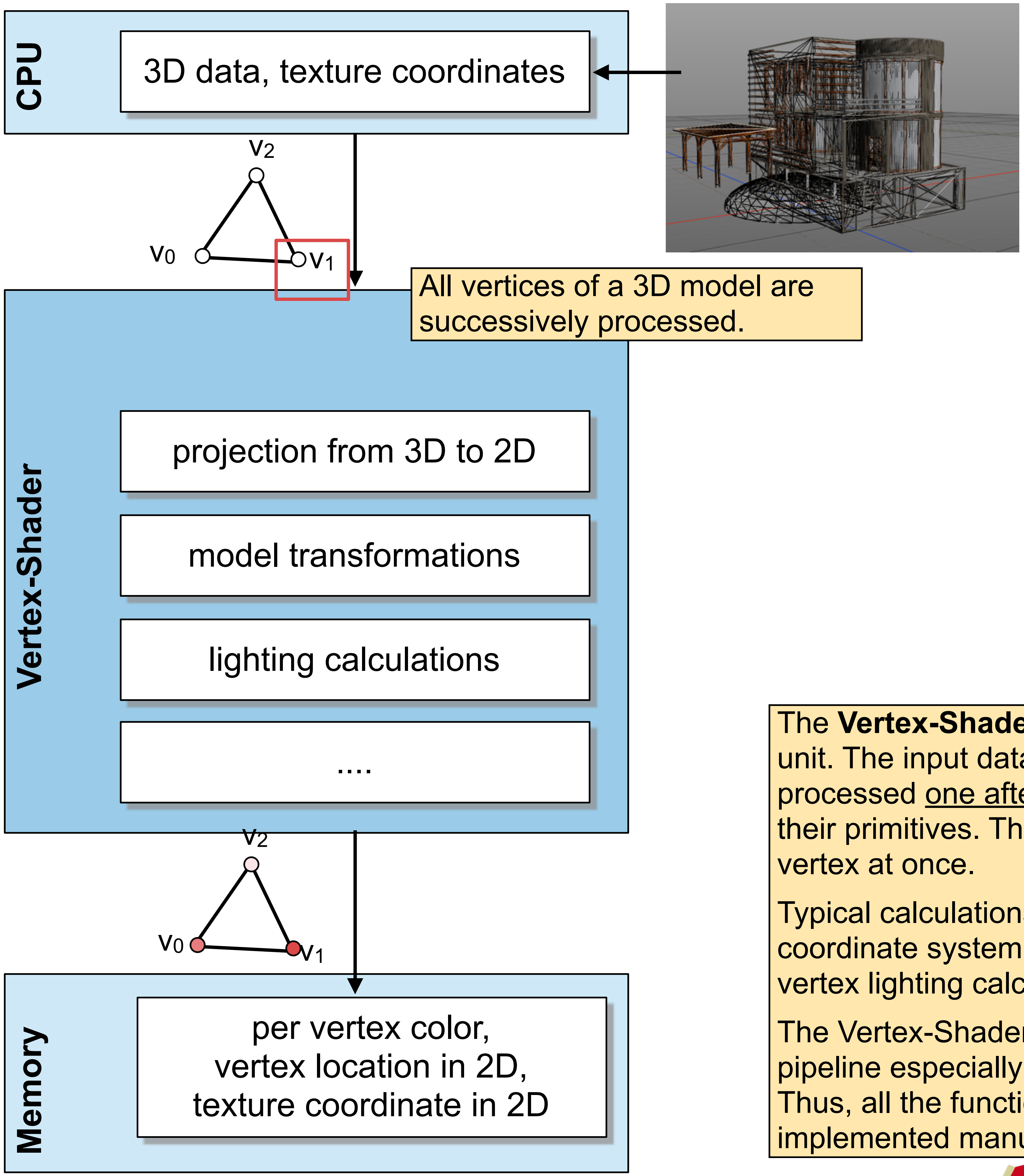
The **programmable rendering pipeline** uses free-programmable logic processors. Thus, they can be used to implement a vast amount of visual effects which goes beyond the capabilities of the fixed function rendering pipeline. The programmer can decide on his/her own, which function need to be implement to realize a distinct effect.

The **Vertex-Shader** is used to manipulate the vertices of a 3D model and to carry out per-vertex lighting calculations.

The **Pixel (Fragment)-Shader** facilitates the manipulation of single pixels.

The **Geometry-Shader** enables the programmer to create new vertices and

Vertex-Shader

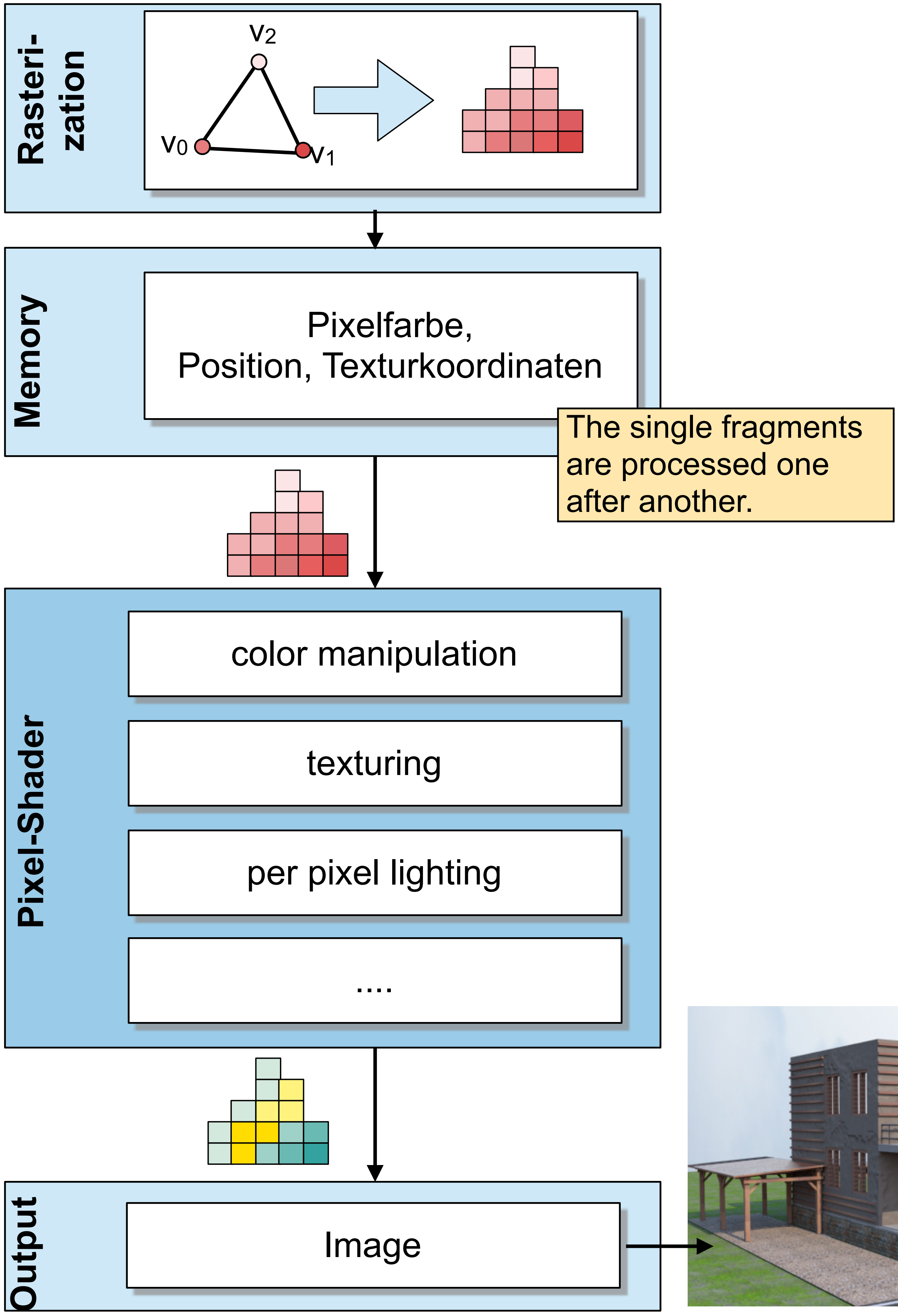


The **Vertex-Shader** serves as a geometry manipulation unit. The input data are all vertices of a 3D model. The are processed <u>one after another</u> and are combined according their primitives. Thus, a Vertex-Shader processes one vertex at once.

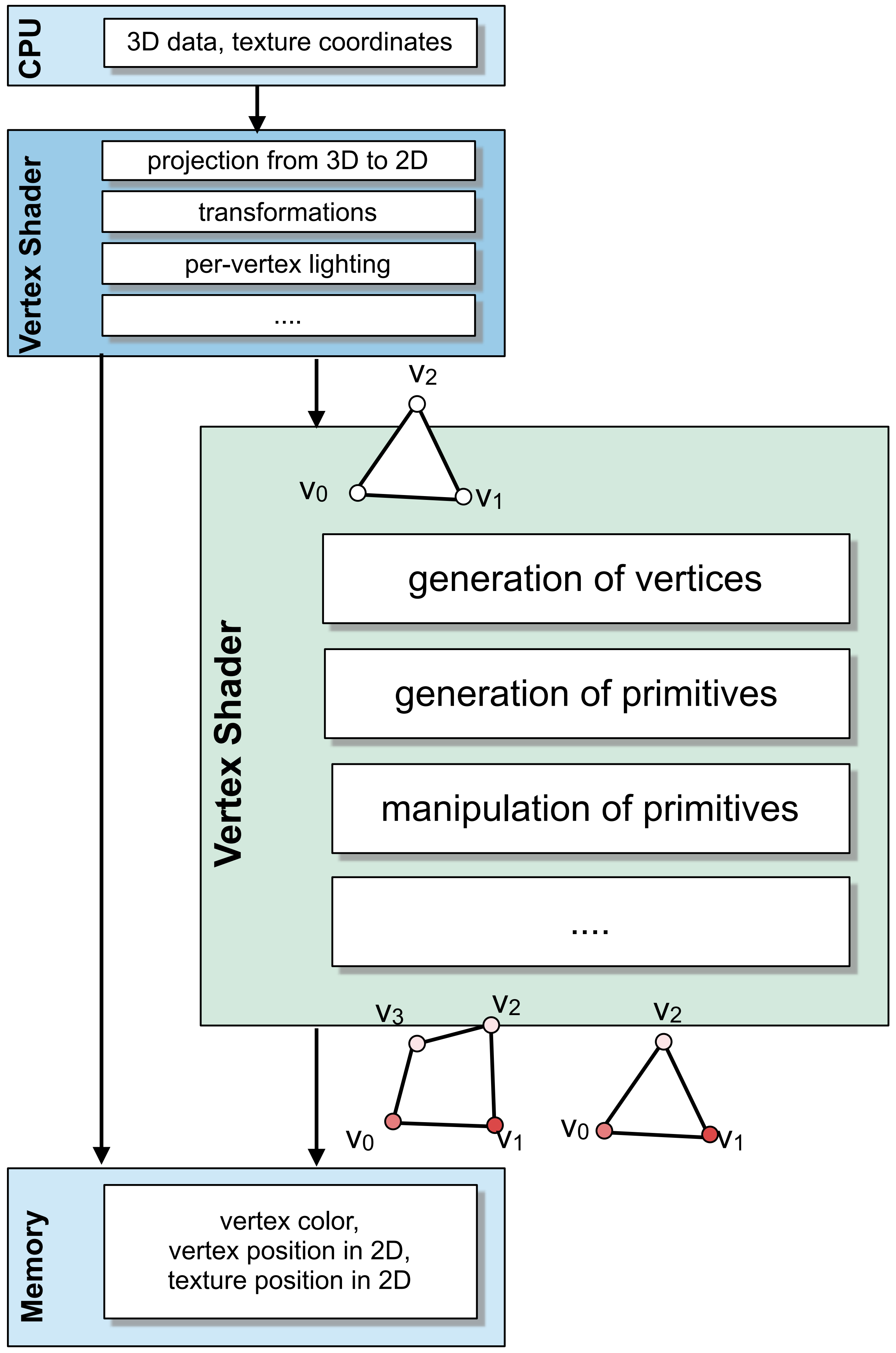
Typical calculations are the projection from 3D to 2D coordinate system, transformation calculations, and per-vertex lighting calculations.

The Vertex-Shader replaces the fixed function rendering pipeline especially the Transformation & Lighting Unit. Thus, all the functions provided by this unit must be implemented manually.

Pixel (Fragment)-Shader



The **Pixel-Shader** (or Fragment-Shader) is used to manipulate the color data of each primitive's fragment that appears on screen. It replace the **Texture Unit** of the fixed function pipeline. Nevertheless, is can be used for a vast amount of additional functions that go beyond the capabilities of the fixed function pipeline.



Geometry-Shader

The **Geometry-Shader** is used for the generation of new vertices and primitives as well as for the manipulation (e.g., Quad to Polygon) of already existing primitives. It is invoked after the Vertex-Shader processing.

Common usage includes i.e., shadow processing and morphing.

Today's realizations are still not performant. Thus, it can only be used to create and manipulate a limited number of vertices and primitives.

Shader Programming Languages

Shading Language

OpenGL Shanding Language (GLSL or glSlang) is a system-independent shader programming language for programmable rendering pipelines. It is similar to C and OpenGL.

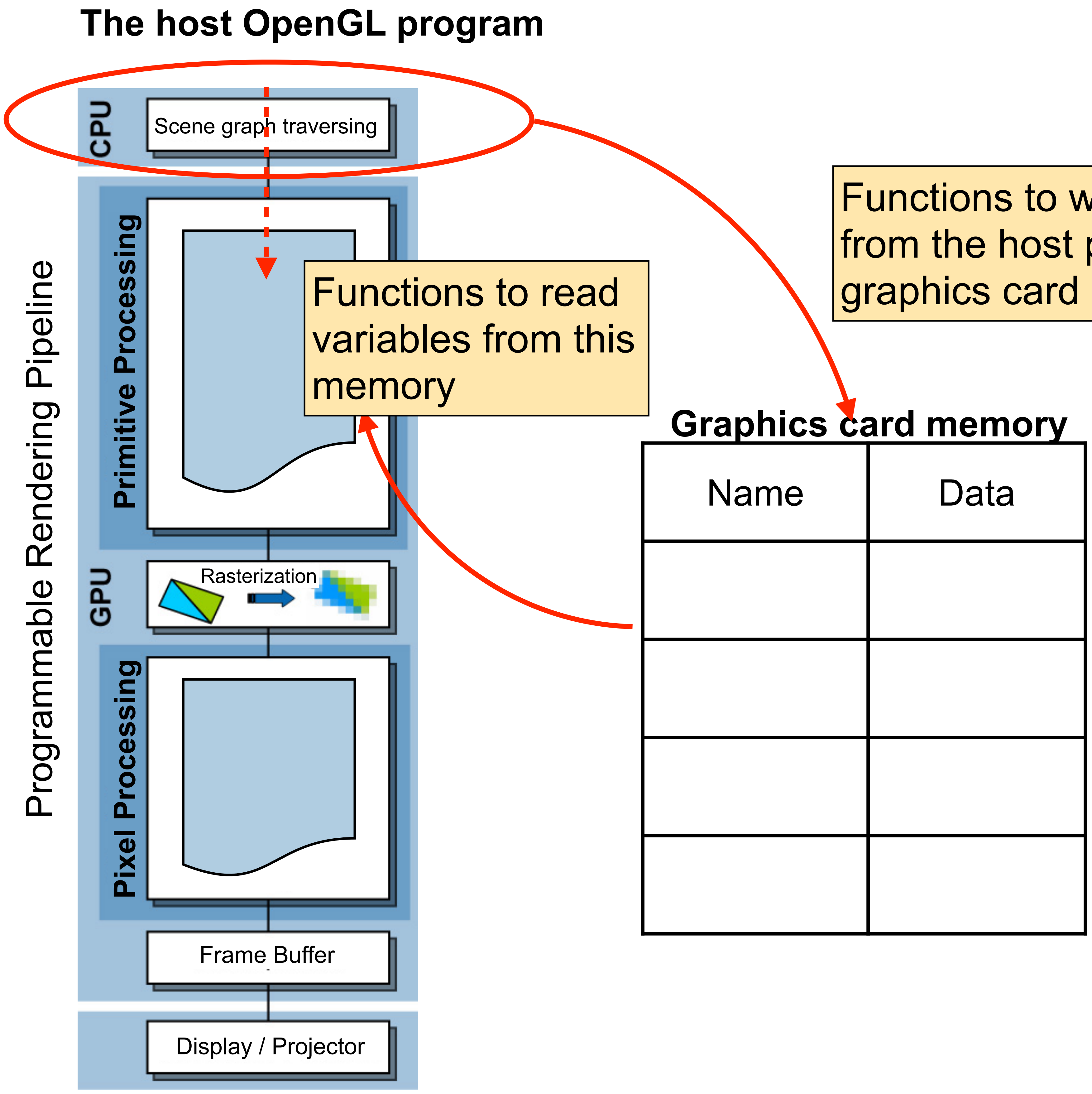
The latest version is GLSL 4.3

C for Graphics (Cg) is a high-level shader programming language. The language works independently from the underlying graphics subsystem; it supports Direct X as well as OpenGL. It has be published by NVIDIA, nevertheless, it also works on ATI Graphics Boards.

NVIDIA offers the Cg Toolkit for a convenient shader development.

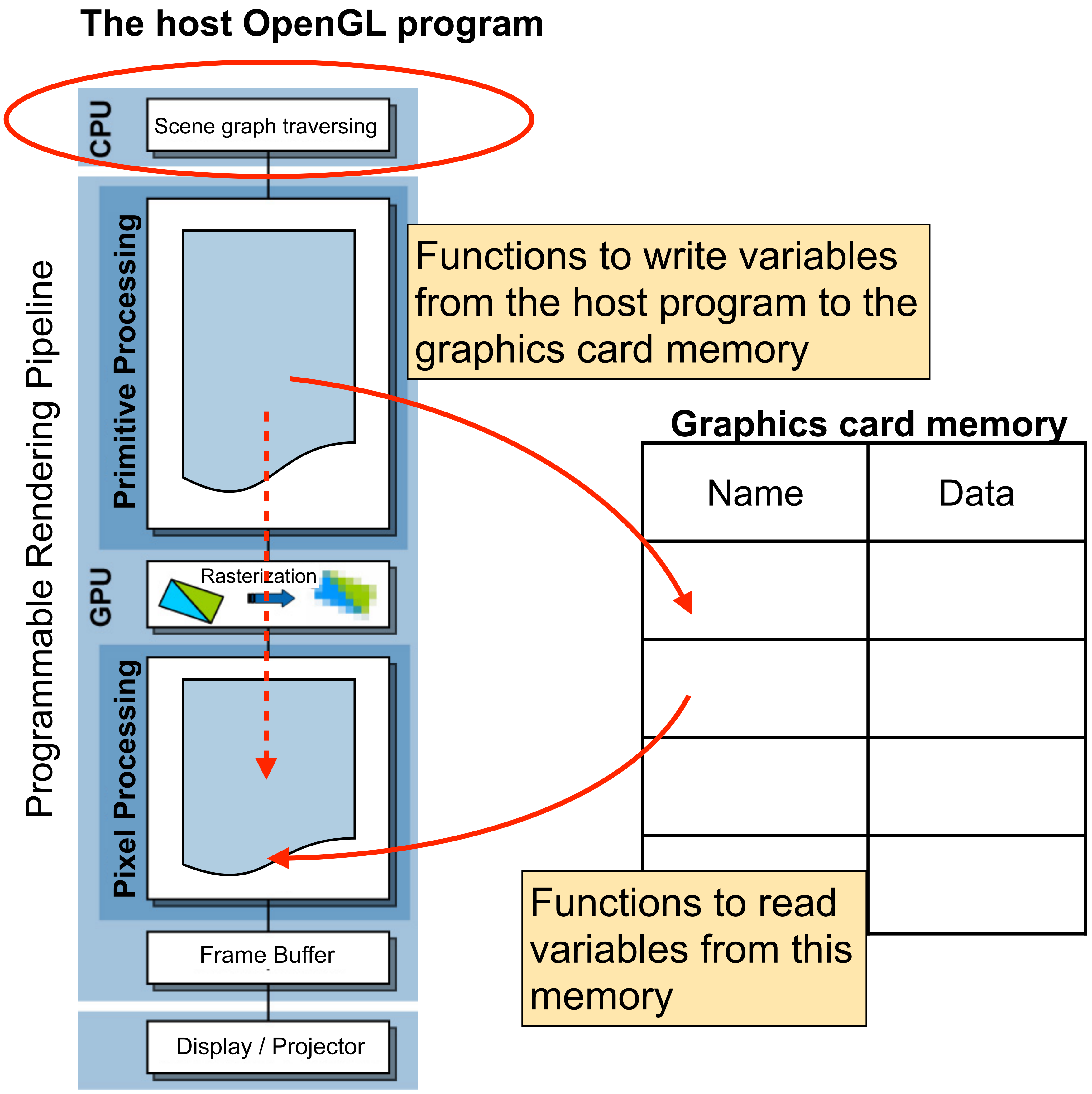
High Level Shading Language (HLSL) is a high-level shader language, published by Microsoft. It is a part of DirectX and conceptually integrated into the Microsoft DirectX programming pattern. Thus, it can only be used within DirectX graphics applications.

Types of Functions / Variables

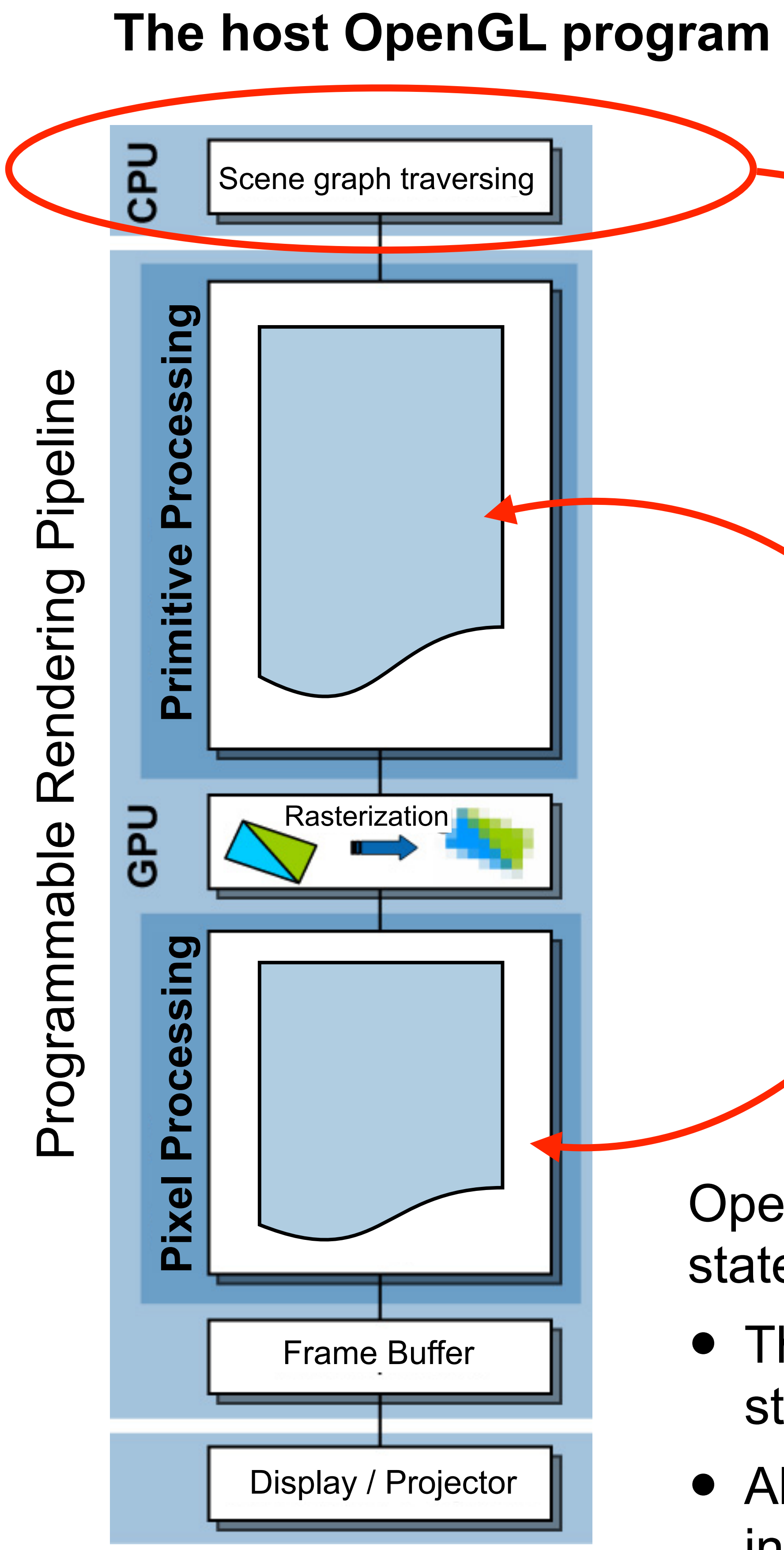

The host OpenGL program
CPU
Scene graph traversing
Programmable Rendering Pipeline
Primitive Processing
GPU
Rasterization
Pixel Processing
Frame Buffer
Display / Projector
Functions to read variables from this memory
Functions to write variables from the host program to the graphics card memory
Graphics card memory

Name	Data

Types of Functions / Variables



State Machine



Graphics card memory

Name	Data
predefined	
predefined	
predefined	
predefined	

OpenGL uses the computational concept of a state machine:

- The graphics card can be set to a certain state (color, transformation, etc.)
- All functions and parameters that are active in a state are used to modify the vertices.

GL Shader Languange

```
uniform vec4 VariableA;
float VariableB;
vec3  VariableC;
const float KonstanteA = 256.0;

float MyFunction(vec4 ArgumentA)
{
    float FunktionsVariableA = float(5.0);

    return float(ArgumentA * (FunktionsVariableA + KonstanteA));
}



// Ich bin ein Kommentar
/* Und ich auch */
void main(void)
{
   gl_Position     = gl_ModelViewProjectionMatrix * gl_Vertex;
   gl_TexCoord[0]  = gl_MultiTexCoord0;
}
```

Declaration of a variable

A function definition looks similar to C/ C++ function definitions.

The entry-point for every vertex and fragement shader program is the function called main.

All variables and functions starting with **gl_*** are so-called built-in variables and functions. Using this variables, a programmer gains access to the data of the rendering pipeline.

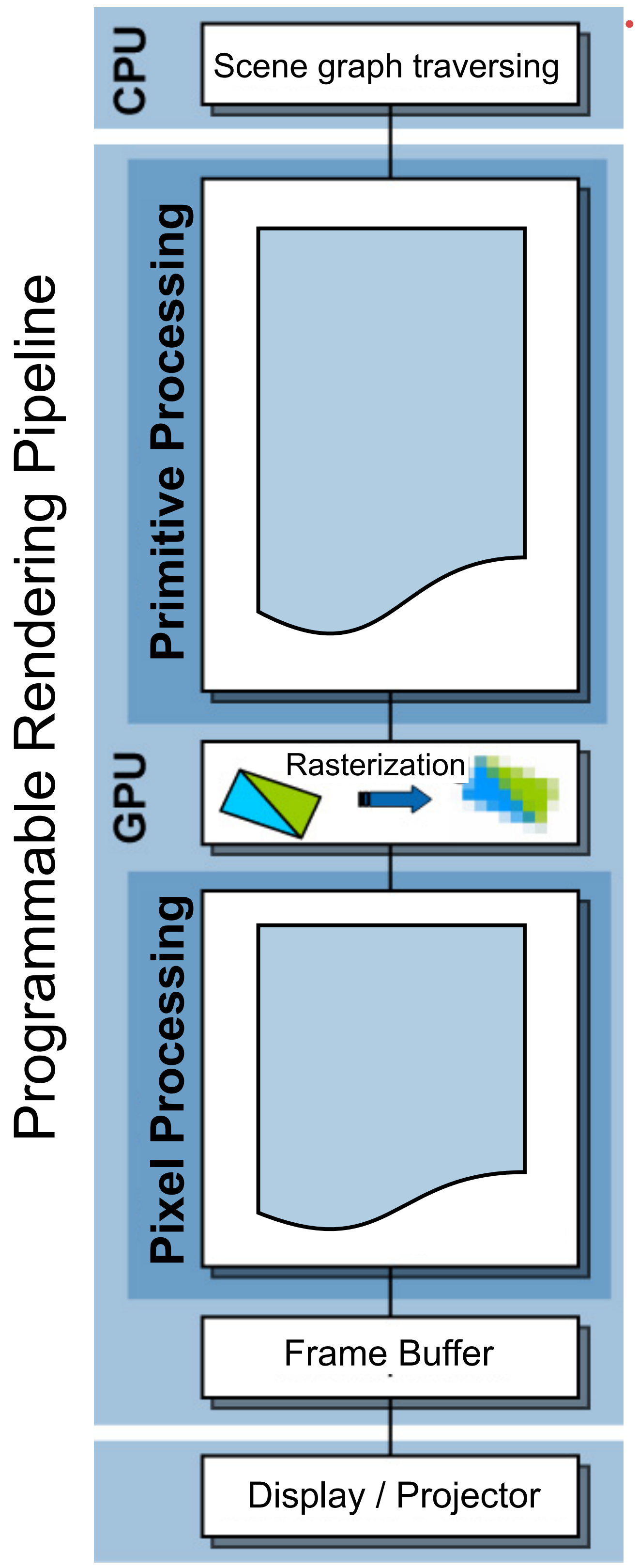
Datatypes

ARLAB

Datentyp	Erklärung
void	For functions, which do not return a value.
bool	conditional type that can be true or false.
int	signed integer
float	floating point value with single precision (32 Bit)
vec2	2-components floating point vector
vec3	3 -components floating point vector
vec4	4 -components floating point vector
bvec2	2-components boolean vector
bvec3	3 -components boolean vector
bvec4	4 -components boolean vector
ivec2	2 -components integer vector
ivec3	3 -components integer vector
ivec4	4 -components integer vector
mat2	2x2 floating point value matrix
mat3	3x3 floating point value matrix
mat4	4x4 floating point value matrix

Datentyp	Erklärung
sampler1D	A 1D data array (a 1D texture).
sampler2D	A 2D data array (a 2D texture).
sampler3D	A 3D data array (a 3D texture).
samplerCube	A cube map texture.
sampler2DRect	Texture which size is not a power of 2 (2^n * 2^n)
sampler1DShadow	A 1D data array (a 1D texture) with an additional comparison operator for shadow textures.
sampler2DShadow	A 2D data array (a 2D texture) with an additional comparison operator for shadow textures.
samplerCubeShadow	A cube map texture with an additional comparison operator for shadow textures (e.g., for omni-directional
sampler2DRectShadow	A 2D shadow texture for 2D-non-power-of-two (NPOT)-textures
sampler1DArray	An array of 1D textures
sampler2DArray	An array of 2D textures
sampler1DArrayShadow	A array of 1D data (a 1D texture) with an additional comparison operator for shadow textures.
sampler2DArrayShadow	A array of 2D data (a 2D texture) with an additional comparison operator for shadow textures.
samplerBuffer	An 1D temporary buffer that can be used as buffer storage
sampler2DMS	A 2D data array (a 2D texture) eligible for multi texturing.
sampler2DMSArray	An arran of 2D data arrays (a 2D textures) eligible for multi texturing.

Type Qualifiers (1/2) uniform and varying



The data type **uniform** allows to pass data from a 3D application to the vertex and fragment shader program.

```
uniform vec4 VariableA;
varying vec3  VariableB;
const float KonstanteA = 256.0;

/* Vertex Programm */
void main(void)
{
   VariableB = vec3(12.0, 13.0, 14.0);
   gl_Position      = gl_ModelViewProjectionMatrix * gl_Vertex;
   gl_TexCoord[0]   = gl_MultiTexCoord0 + VariableA;
}
```

The data type **varying** allows push data from vertex shader to the fragment shader program.

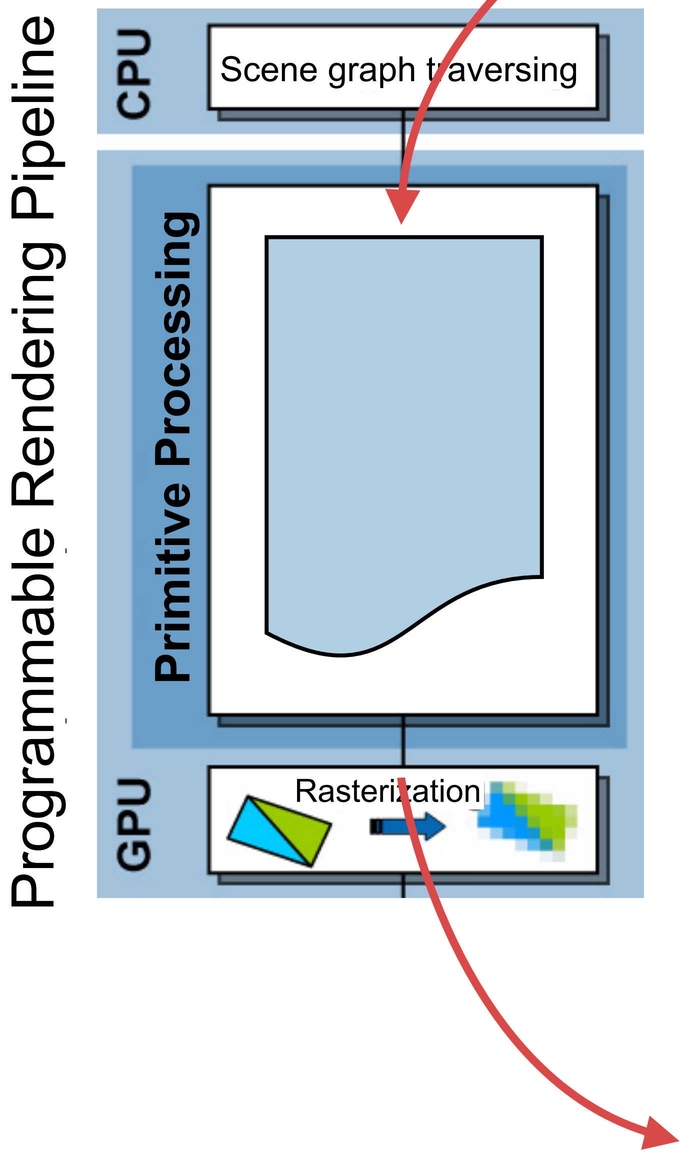
```
varying vec3  VariableB;
uniform vec4 VariableA;
const float KonstanteB = 63.0;

/* Fragment Programm */
void main(void)
{
   gl_FragColor  = gl_FrontLightModelProduct.sceneColor;
}
```

Type Qualifiers (1/2) uniform and varying

In addition to a variable's type, it can bear a Type Qualifier that describes the accessibility of the data.

- **const**
  Fixed (read-only) constant values, available only in the program in which the variable is declared.
- **uniform**
  A variable that poses an interface between the graphics application and the shader program. It can be written and altered within the graphics application and passed to the shader program. The value of the variable can only be changed once at each rendering pass.
- **attribute**
  Read-only, and built-in data that grants access to data of the fixed function rendering pipeline (vertices, color, depth). They pose a pre-defined interface between the graphics application and the shader program.
- **varying**
  Data of type varying are for data exchange between vertex shader program and fragment shader program. They are writeable inside the vertex shader program and read-only in the fragment shader program.
- **in**
  Function variable: an input variable for a distinct function.
- **out**
  Function variable: an output variable for a distinct function.
- **inout**
  Function variable: an input and output variable for a distinct function. It works like a C++ pointer or reference.

Built-in variables and attributes (1/2)



The rendering pipeline automatically provides these attributes; they are only readable; write access is not permitted.

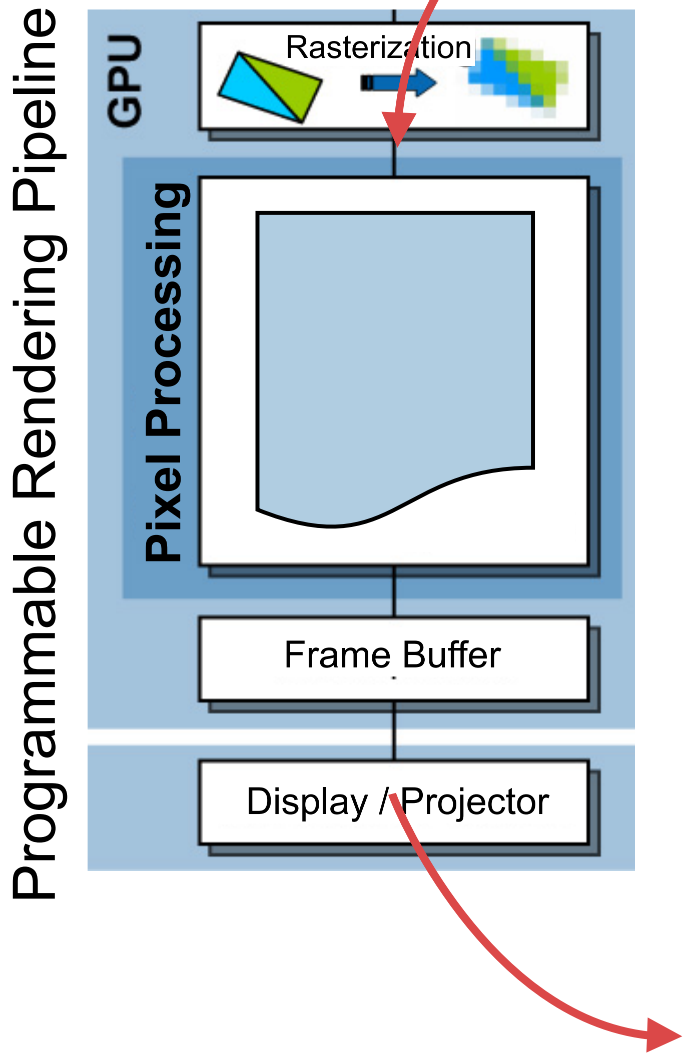
- vec4 gl_Color: Vertex color of this vertex, if provided.
- vec4 gl_SecondaryColor: Secondary certex color of this vertex, if provided.
- vec4 gl_Normal: Vertex normal vector
- vec4 gl_Vertex: Position of the vertex in 3D object coordinate system.
- float gl_FogCoord Fog coordinates

```
uniform vec4 VariableA;
varying vec3  VariableB;
const float KonstanteA = 256.0;

/* Vertex Programm */
void main(void)
{
   VariableB = vec3(12.0, 13.0, 14.0);
   gl_Position     = gl_ModelViewProjectionMatrix * gl_Vertex;
   gl_TexCoord[0]  = gl_MultiTexCoord0;
}
```

- **vec4 gl_Position** must be provided by the programmer.
  This variable stores the position of a vertex in 3D space. The subsequent rasterization unit requires this value to be able to determine the fragments that are covered by a distinct primitive.
- **vec3 gl_TexCoord[x]**
  This variable stores the texture coordinates associated with a distinct vertex. They must be provided if textures available. x is the texture unit.

Built-in variables and attributes (2/2)

The input values of a fragment shader program are all variable that have been defined in the vertex shader program as well as built-in variables of the graphics pipeline.



```
varying vec3  VariableB;
uniform vec4 VariableA;
const float KonstanteB = 63.0;

/* Fragment Programm */
void main(void)
{
   gl_FragColor  = gl_FrontLightModelProduct.sceneColor;
}
```

The following out variables are built-in variables; the pose the output data of the fragment shader program:

- **vec4 gl_FragColor**
  The variable bears the output color of each fragment. The fragment will disappear on screen, if this variable is not specified.
- **vec4 gl_FragData[0..15]**
  Replaces the gl_FragColor variable, if multipass-rendering is used.
- **float gl_FragDepth**
  The variable stores the depth value that is associated with the FragColor of the current primitive.

At least, a shader program writer must provide a gl_FragColor to get an output on screen.

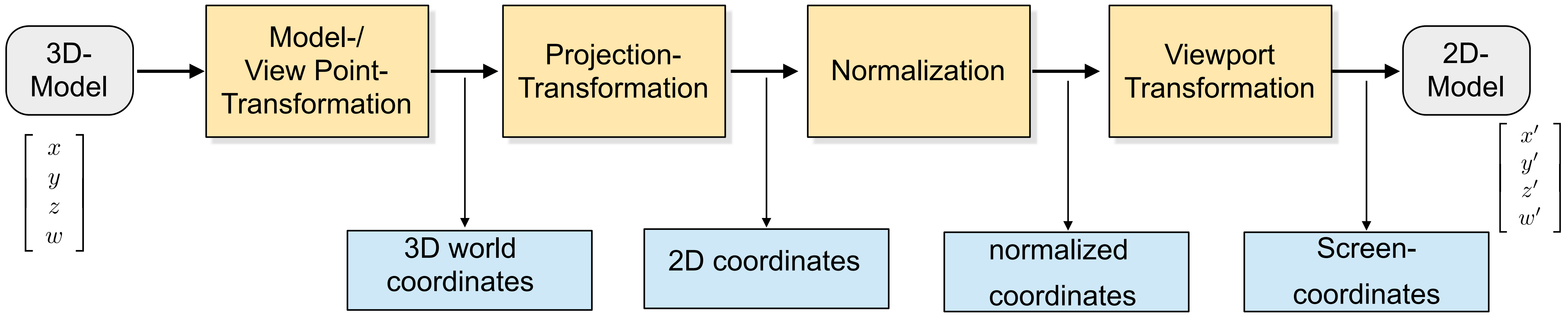
Built-in varyings

All variables of type varying pose a interface between a vertex shader program and a fragment shader program. They are writeable inside a vertex shader and readable inside the fragment shader. The following varyings are built-in varyings provided by OpenGL and GLSL. The can be accessed without being declared manually.

- **vec4 gl_FrontColor, gl_Color**
  Color of the vertex's front
- **vec4 gl_BackColor**
  Color of the vertex's back
- **vec4 gl_FrontSecondaryColor, gl_SecondaryColor**
  Secondary vertex front color
- **vec4 gl_BackSecondaryColor**
  Secondary vertex back color
- **vec4 gl_TexCoord[x]**
  Texture coordinates of the vertex that are stored in texture unit **x**.
- **float gl_FogFragCoord**
  Fog coordinate of that vertex.

Within the fragment shader program, the varying variables `gl_FrontColor`, `gl_FrontSecondaryColor`, `gl_BackColor`, and `gl_BackSecondaryColor` are only accessible via the alias **gl_Color** and **gl_SecondaryColor**, due to the absence of a back color on screen.

Built-in uniforms


3D-Model
$\begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$
Model-/ View Point-Transformation
Projection-Transformation
Normalization
Viewport Transformation
2D-Model
$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix}$
3D world coordinates
2D coordinates
normalized coordinates
Screen-coordinates

Mathematical transformation to transfer each point from 3D space too 2D screen coordinates.

GLSL grants access to typical OpenGL variables commonly used in OpenGL graphics applications.

- **mat4 gl_ModelViewMatrix**
  The model-view-matrix, which describes the transformation from local to global model space.
- **mat4 gl_ProjectionMatrix**
  The projection matrix that poses the projection of the virtual camera
- **mat4 gl_ModelViewProjectionMatrix**
  The product of model-view matrix and projection matrix.
- **mat3 gl_NormalMatrix**
  The normalized matrix, which scales all vertices of a 3D model to a range of [-1,1] and [0,1].
- **mat4 gl_TextureMatrix[gl_MaxTextureCoordsARB]**
  Textures int the texture memory.
- **float gl_NormalScale**
  The OpenGL factor that describes the scaling of surface normal vectors. It helps to determine the scale factor of a 3D model.

Example

Vertex-Shader Program

```
uniform vec4 GlobalColor;

void main(void)
{
   gl_Position      = gl_ModelViewProjectionMatrix * gl_Vertex;
   gl_FrontColor    = gl_Color * GlobalColor;
   gl_TexCoord[0]   = gl_MultiTexCoord0;
}
```

This example shows the most smallest programs necessary to get an output on screen.

Fragment Shader-Program

```
uniform sampler2D Texture0;

void main(void)
{
   vec2 TexCoord = vec2( gl_TexCoord[0] );
   vec4 RGB      = texture2D( Texture0, TexCoord );

   gl_FragColor = RGB + gl_Color;
}
```

Questions ?

Thank you!

Questions

Rafael Radkowski, Ph.D.
Iowa State University
Virtual Reality Applications Center
1620 Howe Hall
Ames, Iowa 50011, USA

+1 515.294.5580

rafael@iastate.edu
http://arlab.me.iastate.edu

www.linkedin.com/in/rradkowski

ARLAB

VRAC|HCI

IOWA STATE UNIVERSITY
OF SCIENCE AND TECHNOLOGY