

Automatic generation of wrapper code for video processing functions

Daniel Oliveira Dantas

Junior Barrera

2/9/2011

As GPU

- GPU = *Graphics Processing Unit*
 - NVIDIA GeForce GTX 460
 - ATI Radeon HD6870
- É um tipo de co-processador com instruções e memória própria

CPU x GPU

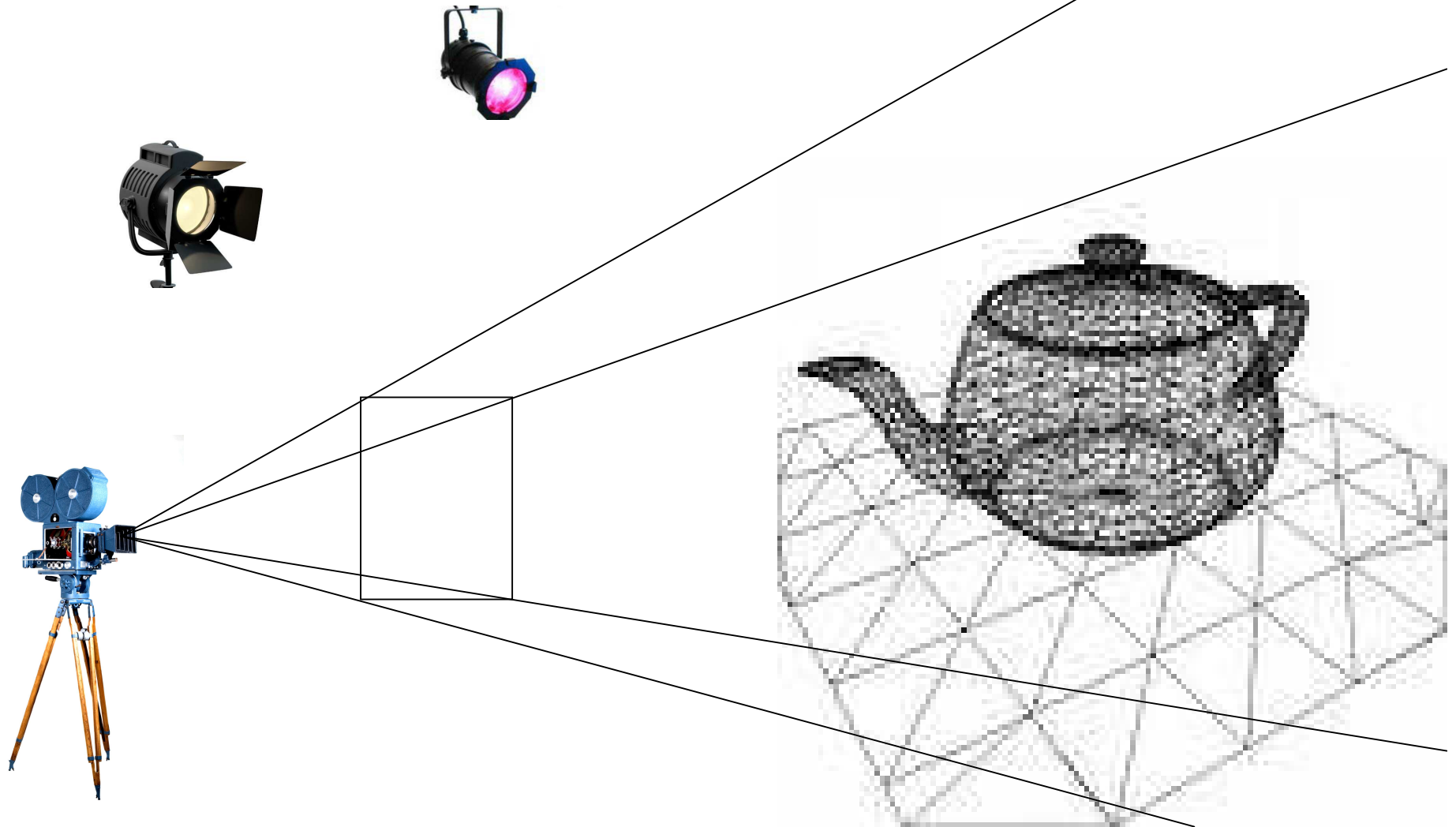
- Arquitetura de Von Neumann
 - SISD
 - Uso genérico: faz de tudo
 - Gargalo entre CPU e memória
 - Estratégias de cache e branch prediction
- Arquitetura paralela
 - SIMD
 - Uso especializado
 - Mais transístores = mais computação

OpenGL

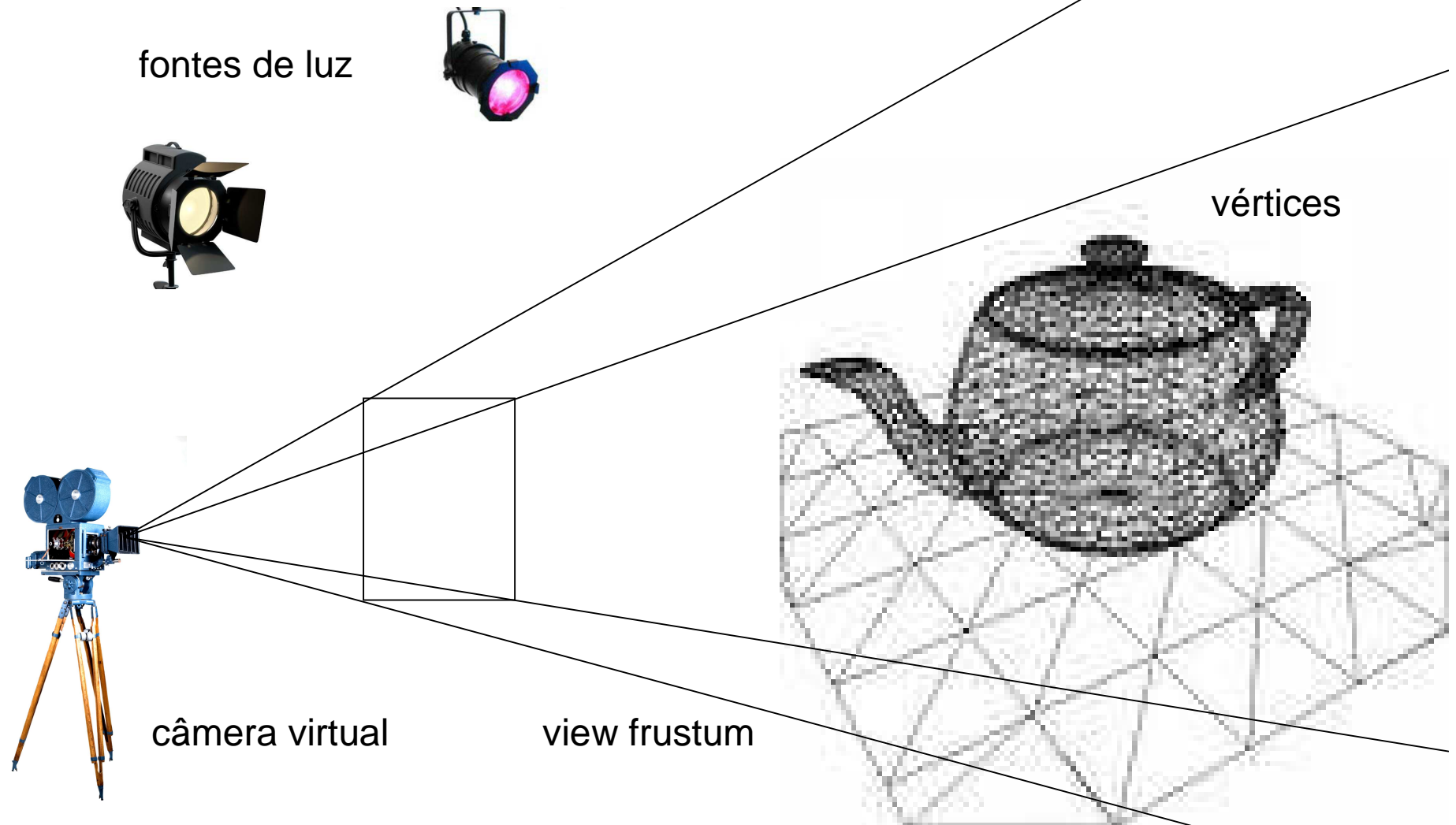
- API gráfica que expõe as funcionalidades das GPU

“OpenGL® is the most widely adopted 2D and 3D graphics API in the industry, bringing thousands of applications to a wide variety of computer platforms. It is window-system and operating-system independent as well as network-transparent. OpenGL enables developers of software for PC, workstation, and supercomputing hardware to create high-performance, visually compelling graphics software applications, in markets such as CAD, content creation, energy, entertainment, game development, manufacturing, medical, and virtual reality. OpenGL 3.1 exposes all the features of the latest graphics hardware. ” (www.khronos.org)

Uma cena OpenGL



Uma cena OpenGL



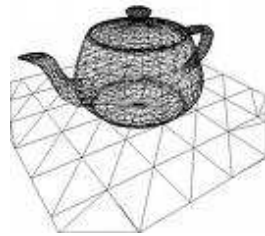
Uma cena OpenGL

posição e
orientação
da câmera



+

vértices



+

texturas



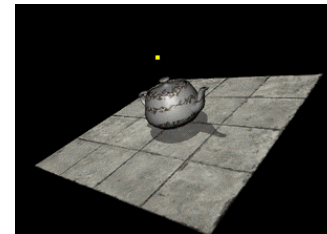
+

posição e
cor das
fontes de luz



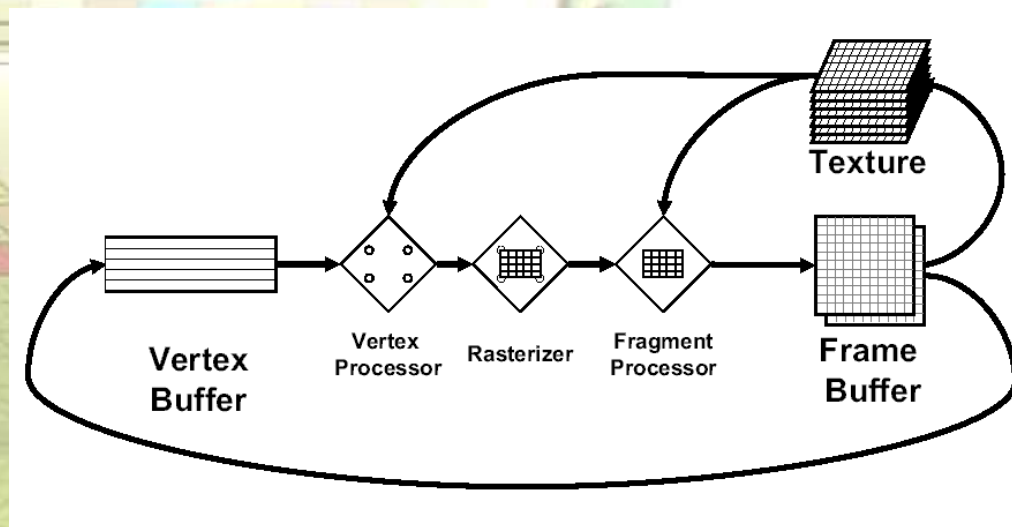
=

imagem

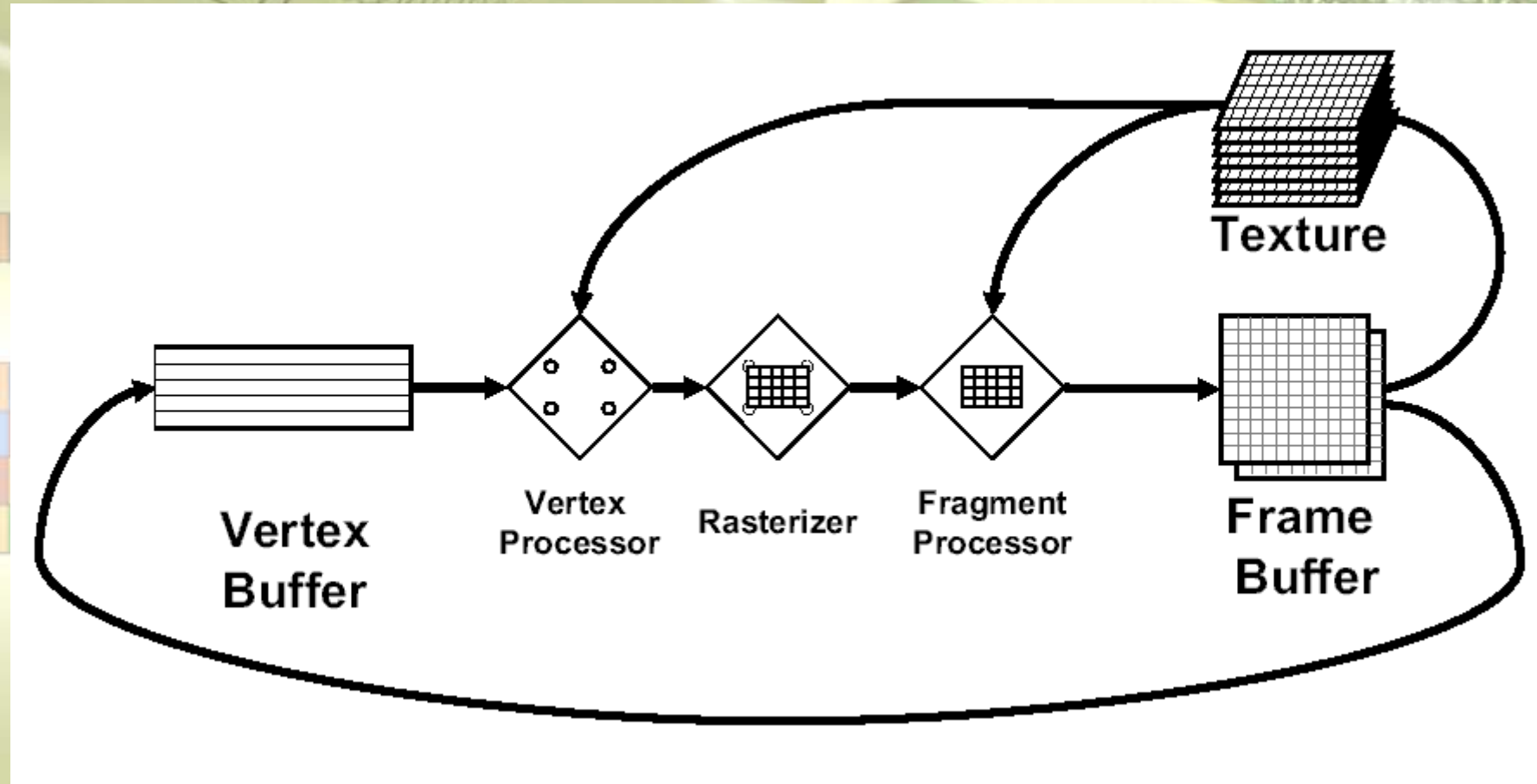


A linguagem GLSL

- Linguagem que faz parte da especificação OpenGL, criada para adicionar flexibilidade ao pipeline fixo de renderização.



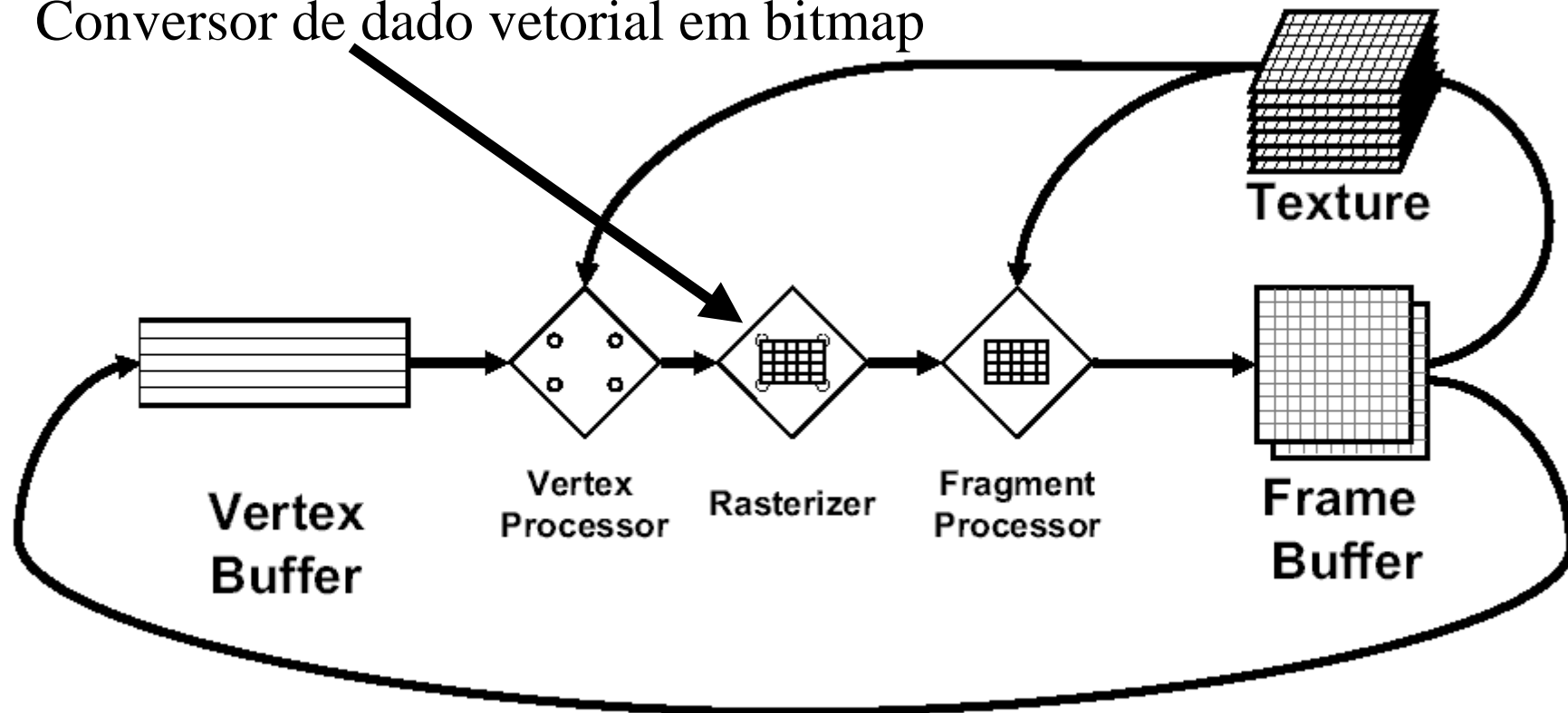
Pipeline OpenGL



- John D. Owens et al, A survey of general-purpose computation on gra hardware, 2005

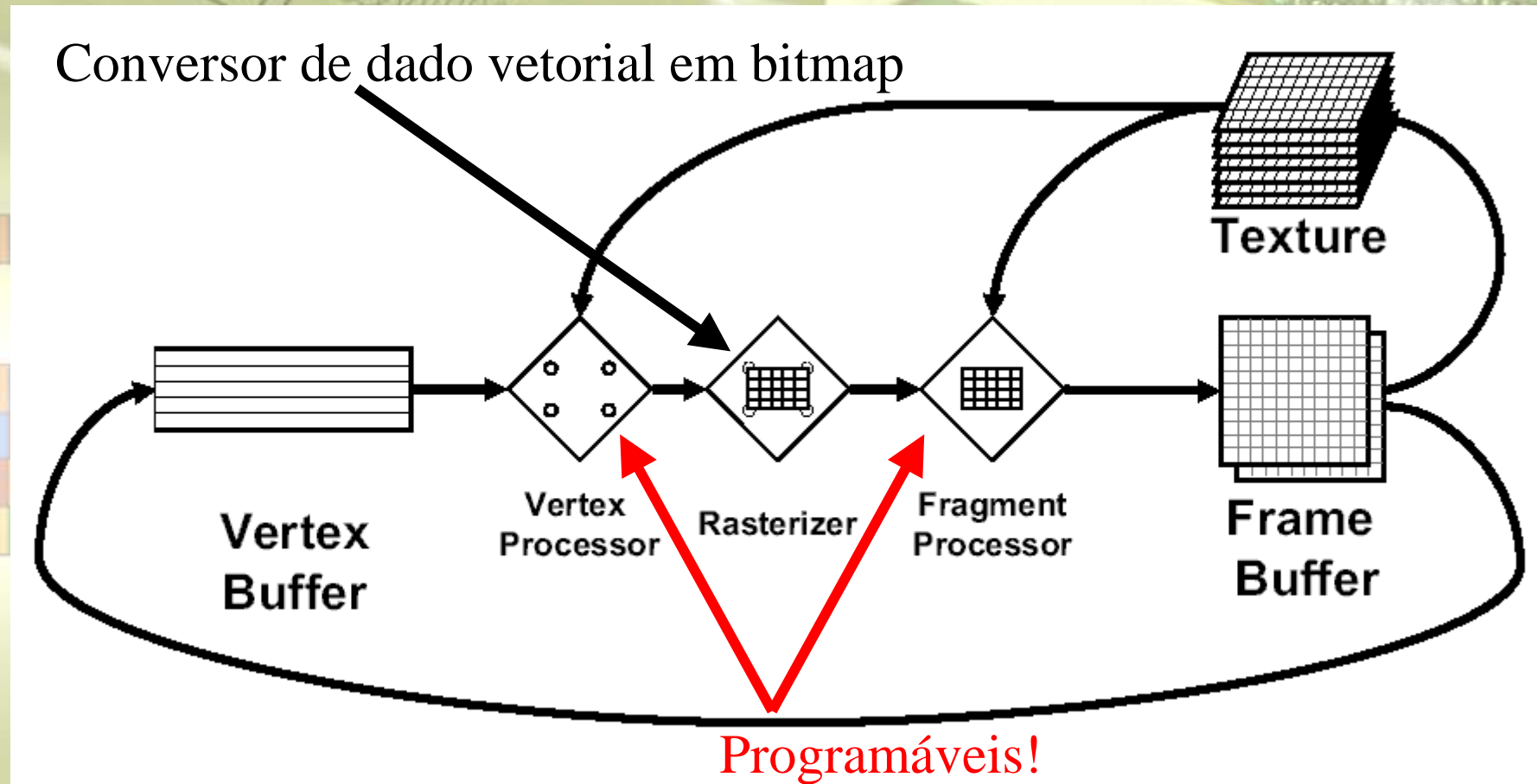
Pipeline OpenGL

Conversor de dado vetorial em bitmap



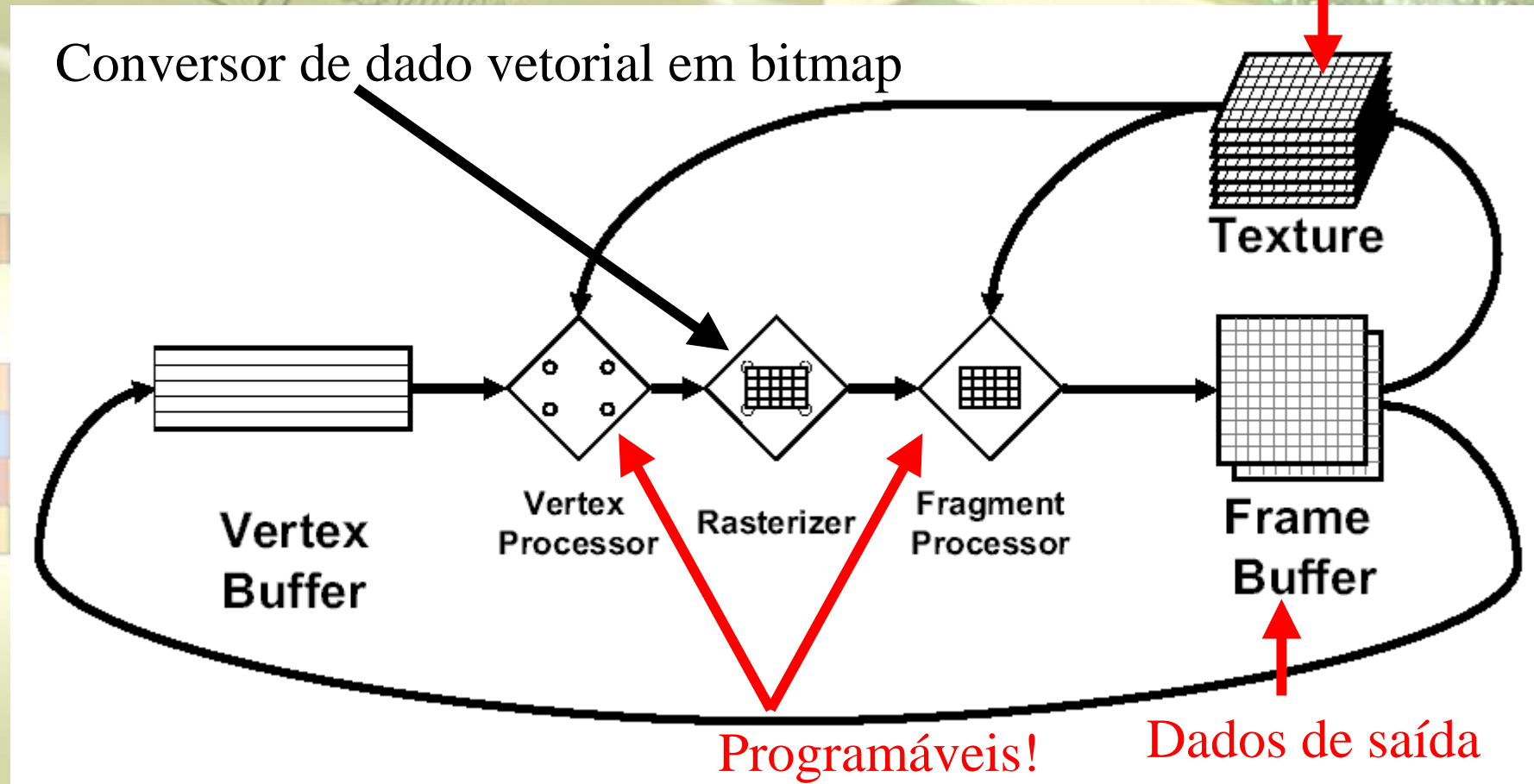
- John D. Owens et al, A survey of general-purpose computation on gra hardware, 2005

Pipeline OpenGL



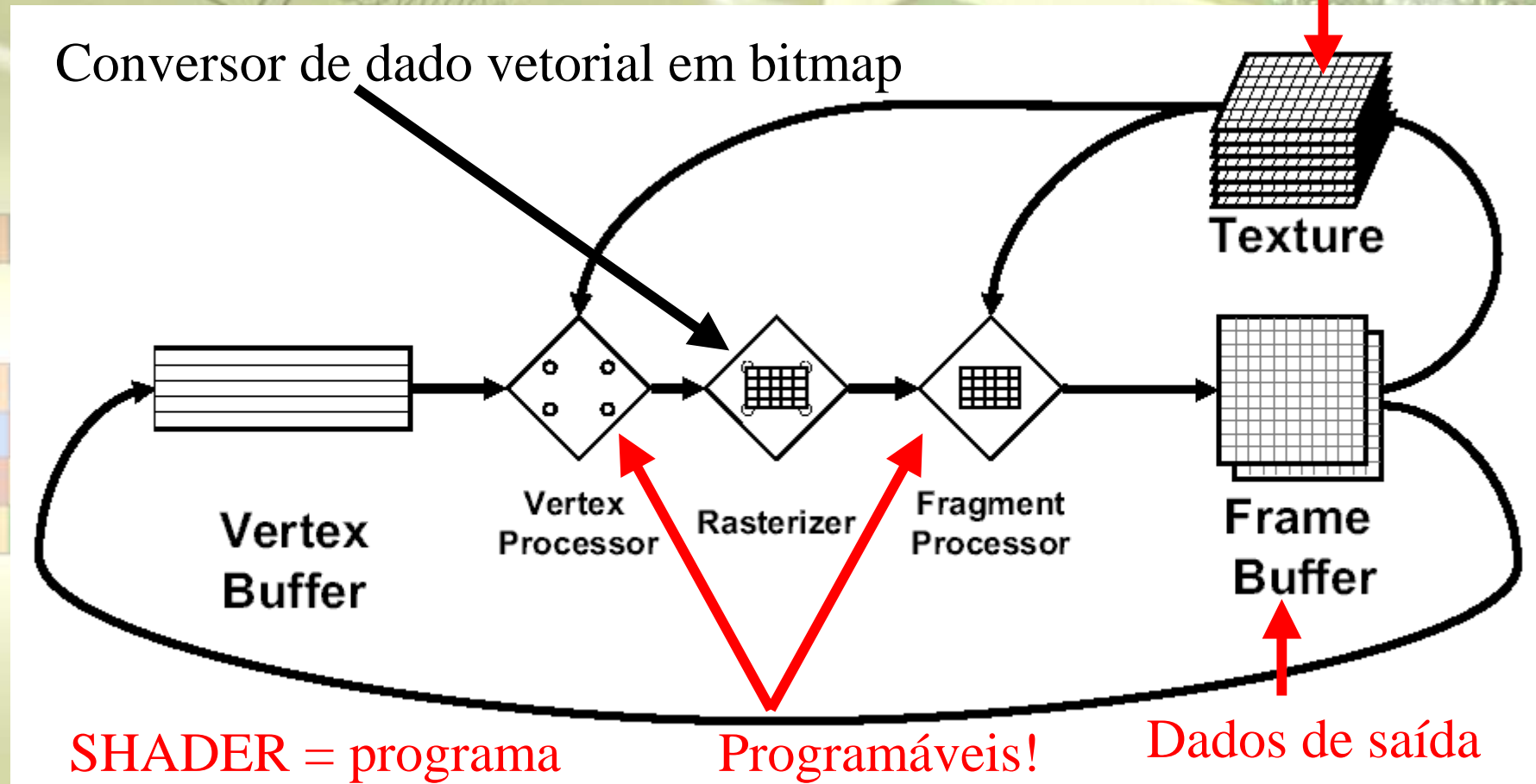
- John D. Owens et al, A survey of general-purpose computation on gra hardware, 2005

Pipeline OpenGL



- John D. Owens et al, A survey of general-purpose computation on gra hardware, 2005

Pipeline OpenGL



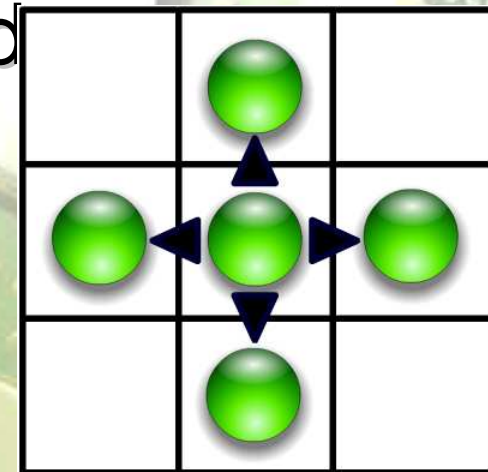
- John D. Owens et al, A survey of general-purpose computation on gra hardware, 2005

Os shaders

- Shader é um programa usado para determinar as propriedades finais da superfície de um objeto.
- Rodam no Vertex processor e Fragment processor
- São equivalentes ao kernel de uma convolução

Vertex shader

- Dados de entrada são triângulos
- Dados de saída são os triângulos transformados. O *rasterizer* os transforma em fragmentos, ou seja, pixels com informação de profundidade
- Operações de scatter: $d[a] = v;$

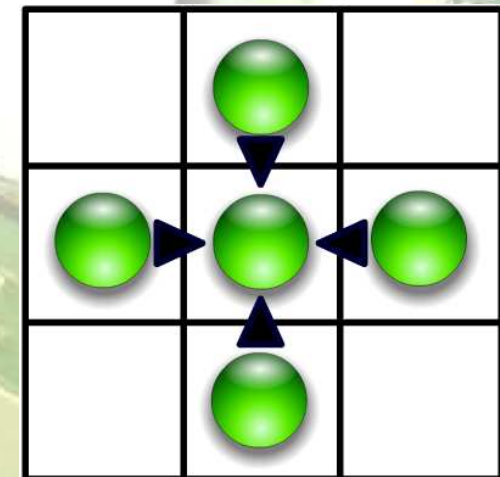


Scatter

Fragment shader

- Dados de entrada são fragmentos
- Dados de saída são pixels

- Operações de gather: $v = d[a]$;



Gather

A linguagem GLSL

- Usada para criar Vertex e Fragment shaders
- Expõe as funcionalidades das GPU
- Facilidade de programação
- C-like syntax
- Portável, expandível
- Permite uso genérico

A linguagem GLSL

```
uniform sampler2D sampler0;
uniform float thresh;
uniform float top;
void main(void){
    vec3 pix = texture2D(sampler0, gl_TexCoord[0].st).rgb;
    if (pix.r > thresh) pix.r = top;
    else pix.r = 0.;
    if (pix.g > thresh) pix.g = top;
    else pix.g = 0.;
    if (pix.b > thresh) pix.b = top;
    else pix.b = 0.;
    gl_FragColor = vec4(pix, 1.);
}
/* LEGENDA
Tipos de dados
Funções
Variáveis
Variáveis embutidas
Qualifiers
*/
```

Tipos de dados

- Básicos
 - `float`: floating point
 - `int`: 16 bit integer
 - `bool`: produzido por comparações e usado nos `if`
 - `sampler*`: para manusear texturas. Não podem ser alteradas pelo shader, apenas fornecidas como parâmetros uniform.
- Vetores
 - `vec2, vec3, vec4`
 - `ivec2, ivec3, ivec4`
 - `bvec2, bvec3, bvec4`
- Matrizes
 - `mat2, mat3, mat4`

Funções embutidas

- Trigonometria
 - `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `radians`, `degrees`.
- Matemática
 - `pow`, `exp`, `log`, `exp2`, `log2`, `sqrt`, `inversesqrt`, `abs`, `sign`, `floor`, `ceil`, `fract`, `mod`, `min`, `max`, `clamp`, `step`, `smoothstep`...
- Geometria
 - `length`, `distance`, `dot`, `cross`, `normalize`, `faceforward`, `reflect`, `refract`...
- Mapeamento de texturas
 - `texture*D`, `texture*DProj`, `texture*DLod`, `textureCube`...
- Outras
 - `dFdx`, `dFdy`, `fwidth`, `matrixCompMult`, `lessThan`, `lessThanEqual`, `equal`...

Variáveis embutidas

- Variáveis de entrada
 - `gl_FragCoord`, `gl_TexCoord`
- Variáveis de saída
 - `gl_FragColor`, `gl_FragData[]`, `gl_FragDepth`
- Matrizes
 - `gl_ModelViewMatrix`, `gl_ProjectionMatrix`,
`gl_ModelViewProjectionMatrix...`
- Outras
 - `gl_ClipPlane[]`, `gl_Point`,
`gl_FrontMaterial`, `gl_BackMaterial`,
`gl_LightSource[]`, `gl_LightModel`, `glFog`

Qualifiers

- `uniform`: dados de entrada cujos valores são iguais para todas as instâncias do shader
- `const`: para declarar variáveis com valor definido em tempo de compilação, como em C

A linguagem CUDA

- Criada para permitir o uso das GPU sem a necessidade de mapear os dados e operações para uma API gráfica
- C-like syntax
- Permite uso genérico



A linguagem CUDA

```
template<typename T>
__global__ void global_Invert(T* input, T*
output, int w, int h, int nChannels){
    int offset = blockIdx.x * nChannels * w;
    T* array_in = input + offset;
    T* array_out = output + offset;
    for (int j = threadIdx.x; j < nChannels *
w; j += blockDim.x){
        array_out[j] = -array_in[j];
    }
}
/* LEGENDA
Tipos de dados
Funções
Variáveis
Variáveis embutidas
Qualifiers
*/
```

Tipos de dados

- Básicos

- `char, uchar`: 8 bit
- `short, ushort`: 16 bit integer
- `int, uint`: 32 bit integer
- `long, ulong`: 64 bit integer
- `float`: floating point
- `Texture<Type, Dim>`: para manusear texturas.

- Vetores

- `dim3`: componentes não especificados recebem 1.
- `char1, ..., char4, uchar1, ..., uchar4`
- `short1, ..., short4, ushort1, ..., ushort4`
- `int1, ..., int4, uint1, ..., uint4`
- `long1, ..., long4, ulong1, ..., ulong4`
- `float1, ..., float4`

Funções embutidas

- Trigonometria

- `sinf`, `cosf`, `tanf`, `asinf`, `acosf`, `atanf`...

- Matemática

- `sqrtf`, `rsqrtf`, `expf`, `exp2f`, `exp10f`, `logf`,
`log2f`, `log10f`, `powf`, `remainderf`, `modff`,
`truncf`, `roundf`, `ceilf`, `floorf`, `lroundf`,
`fminf`, `fmaxf`, `fabsf`...

- Outras

- `isinf`, `isnan`, `isfinite`...

Qualifiers

- De funções

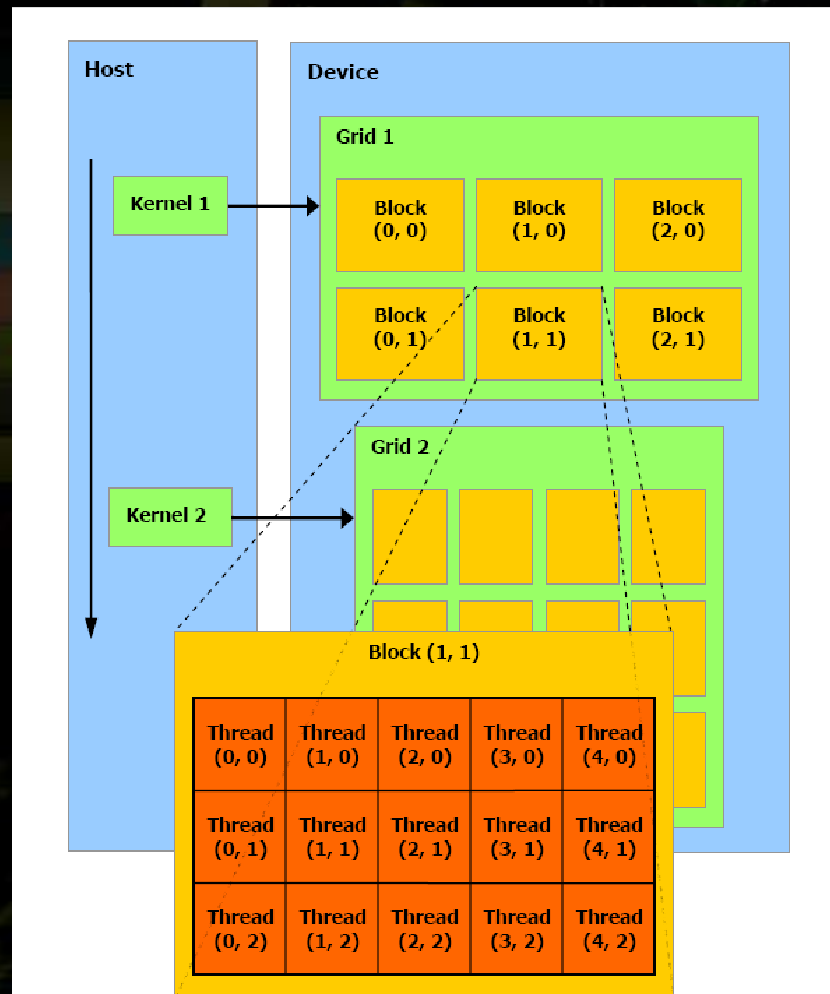
- `__host__`: declara funções que rodam e são chamadas a partir do host (CPU).
- `__device__`: declara funções que rodam e são chamadas a partir do device (GPU).
- `__global__`: declara funções que rodam no device e são chamadas a partir do host.

- De variáveis

- `__device__`: declara variáveis que residem no device (GPU)
- `__constant__`: declara variáveis que residem no device (GPU) e que não são podem ser alteradas.

Configuração de execução

- Toda chamada para funções `__global__` deve especificar uma configuração de execução



Configuração de execução

- Toda chamada para funções `__global__` deve especificar uma configuração de execução

Uma função declarada como

```
__global__ void Func(float f);
```

...deve ser chamada assim:

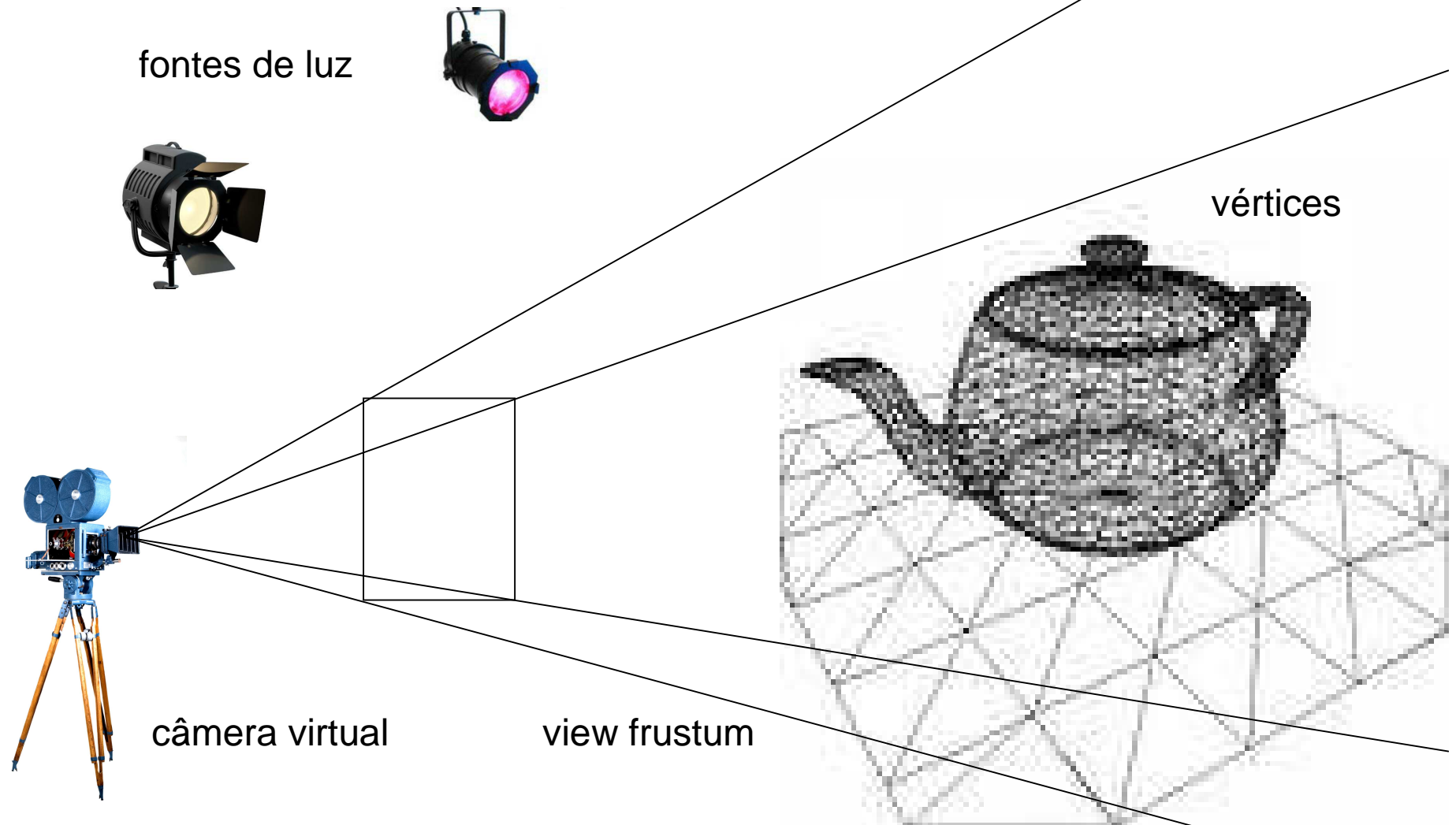
```
Func<<<Dg, Db>>>(f)
```

...onde `Dg` é o tamanho do grid, e `Db` é o tamanho de cada bloco

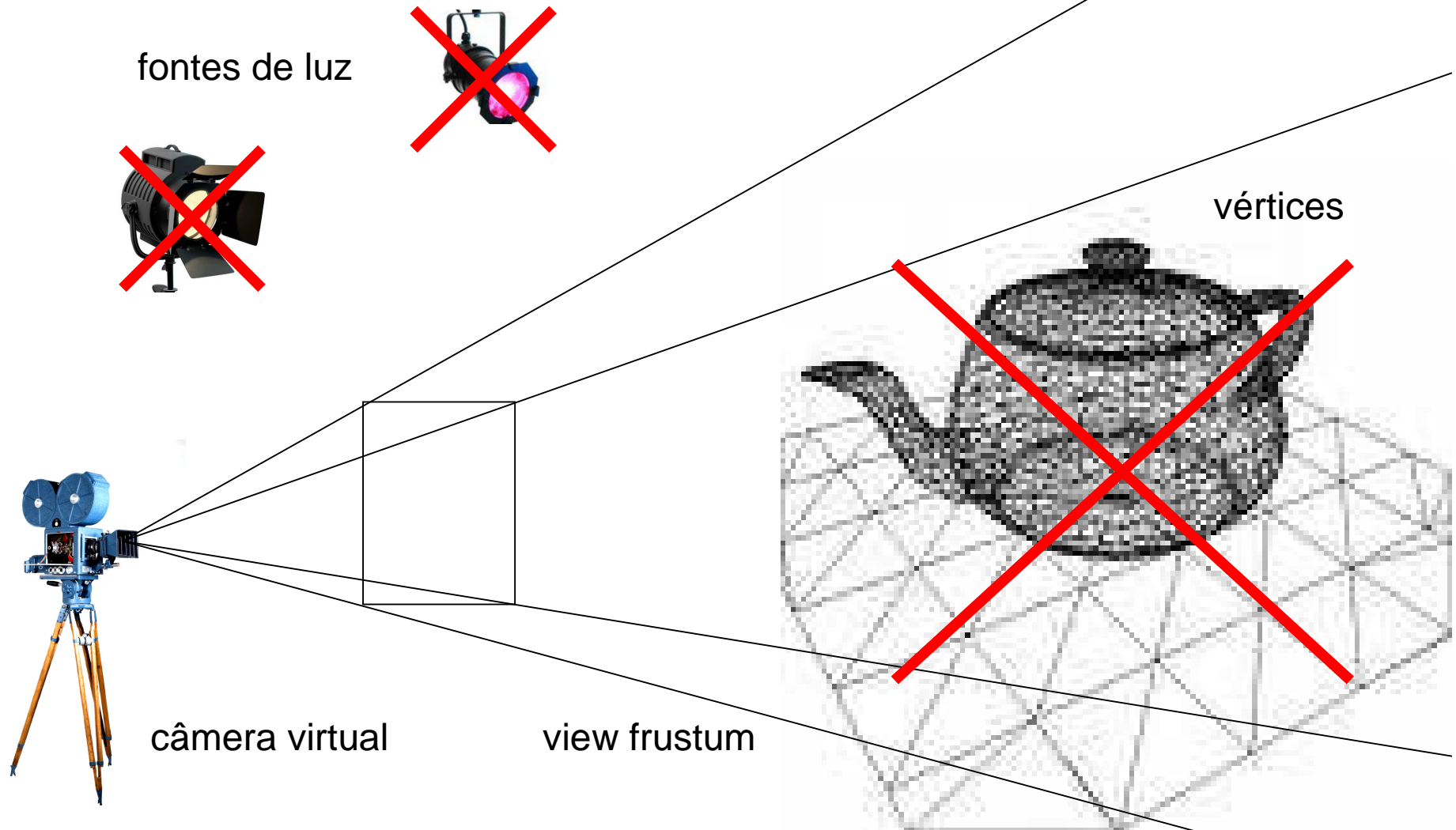
Variáveis embutidas

- Variáveis de entrada:
 - `gridDim`: contém as dimensões do grid
 - `blockIdx`: índice do bloco no grid
 - `blockDim`: contém as dimensões do bloco
 - `threadIdx`: índice da thread no bloco

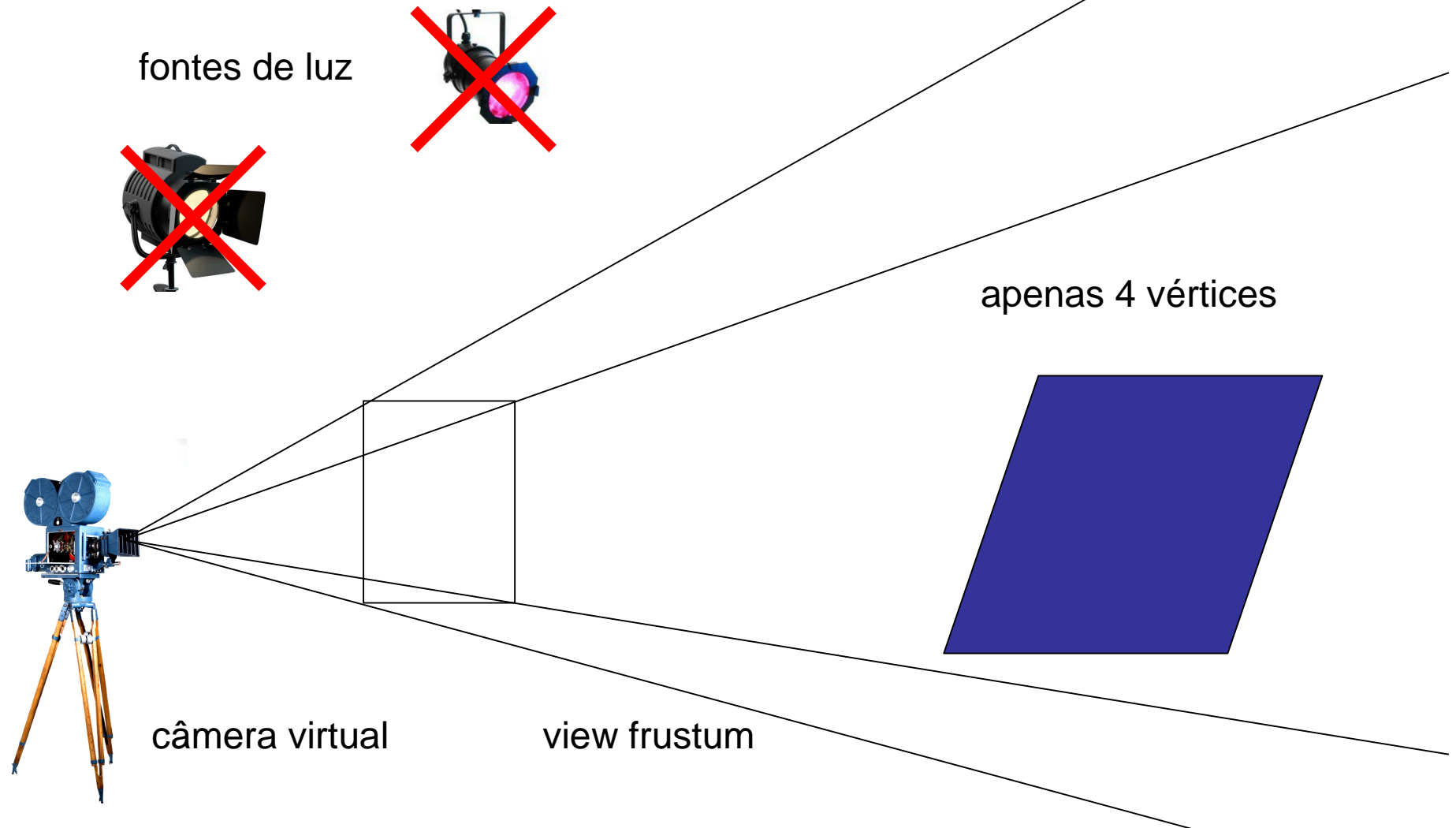
Processando imagens



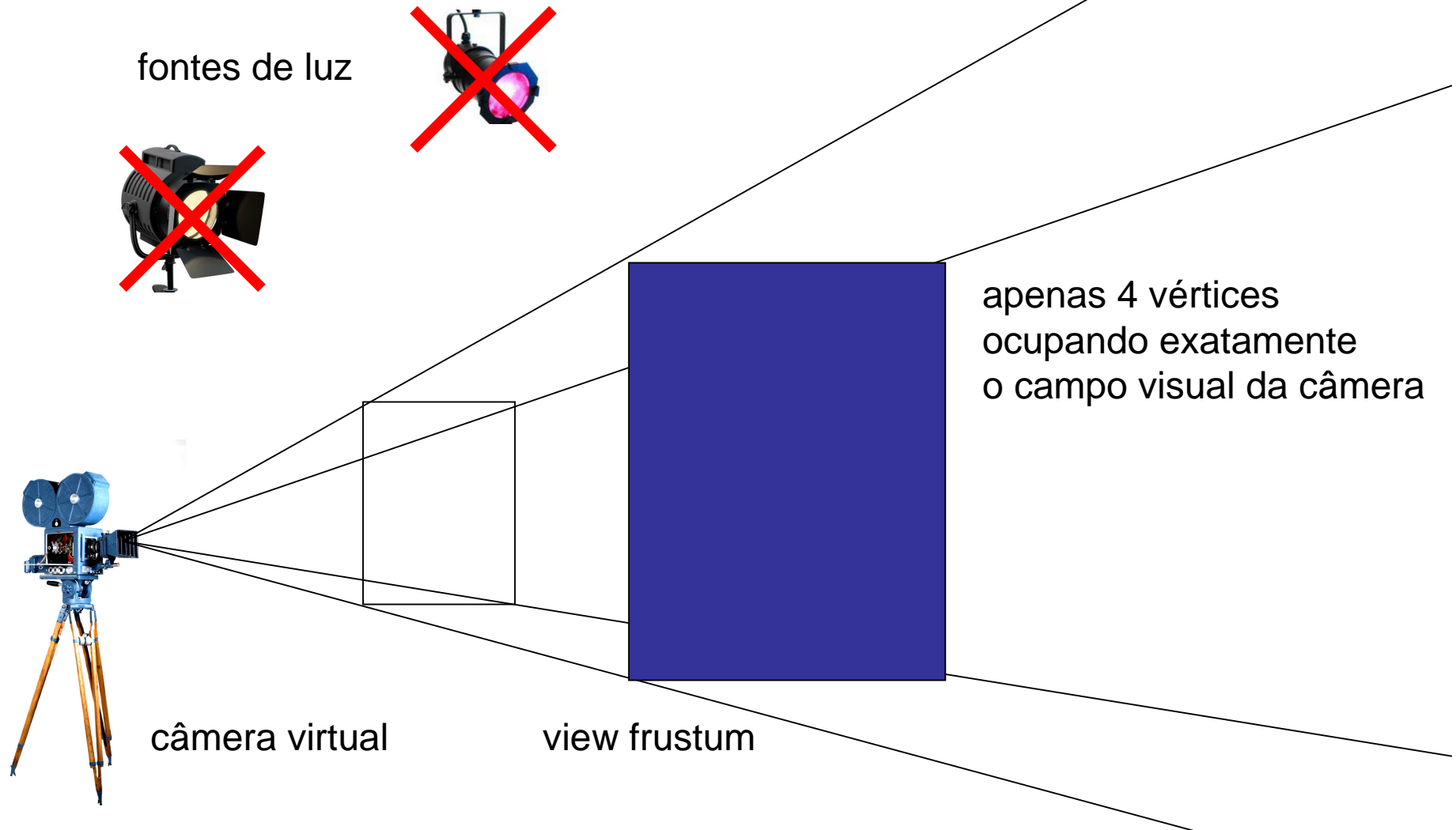
Processando imagens



Processando imagens



Processando imagens

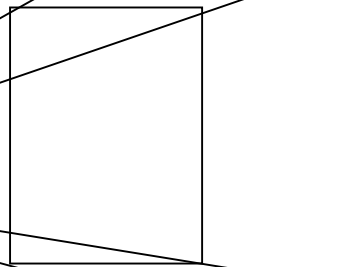


Processando imagens

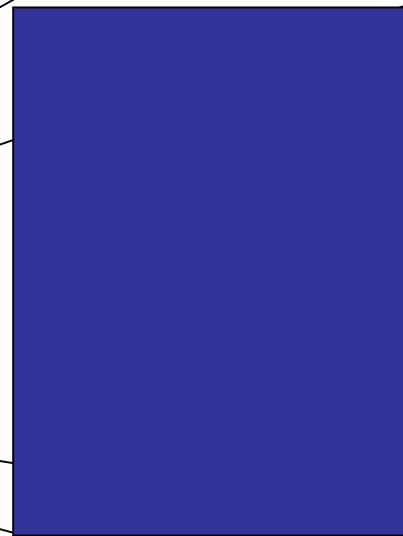
imagem armazenada como textura



câmera virtual



view frustum



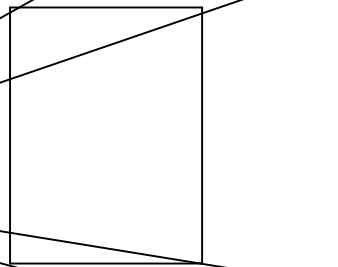
apenas 4 vértices
ocupando exatamente
o campo visual da câmera

Processando imagens

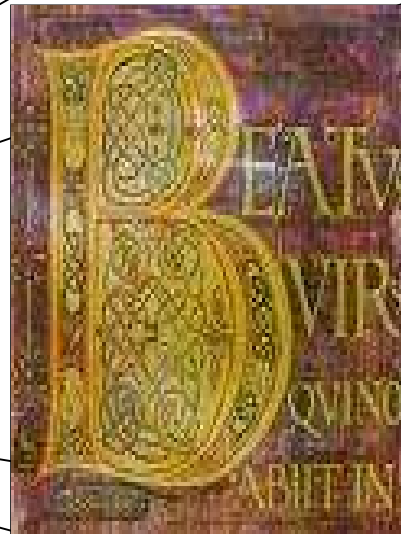
imagem armazenada como textura



câmera virtual



view frustum



apenas 4 vértices
ocupando exatamente
o campo visual da câmera

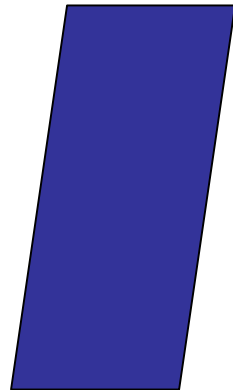
Processando imagens

posição e
orientação
da câmera



+

4 vértices



+

texturas



+

shaders

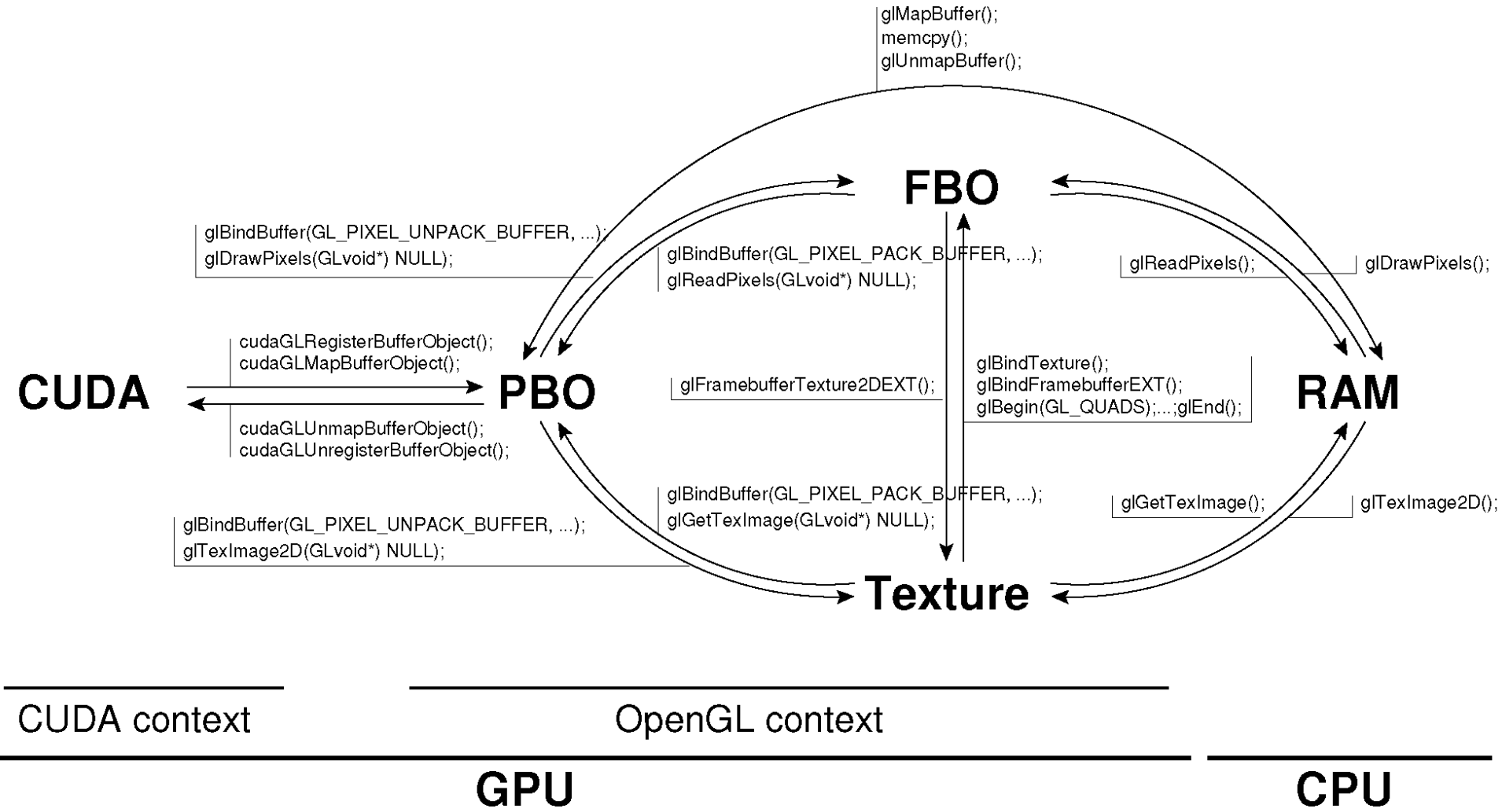
```
uniform sampler2D sampler0;  
uniform float thresh;  
uniform float top;  
void main(void) {  
    vec3 pix = texture2D(sampler0, gl_TexCoord[0].st).rgb;  
    if (pix.r > thresh) pix.r = top;  
    else pix.r = 0.;  
    if (pix.g > thresh) pix.g = top;  
    else pix.g = 0.;  
    if (pix.b > thresh) pix.b = top;  
    else pix.b = 0.;  
    gl_FragColor = vec4(pix, 1.);  
}
```

Imagem de saída

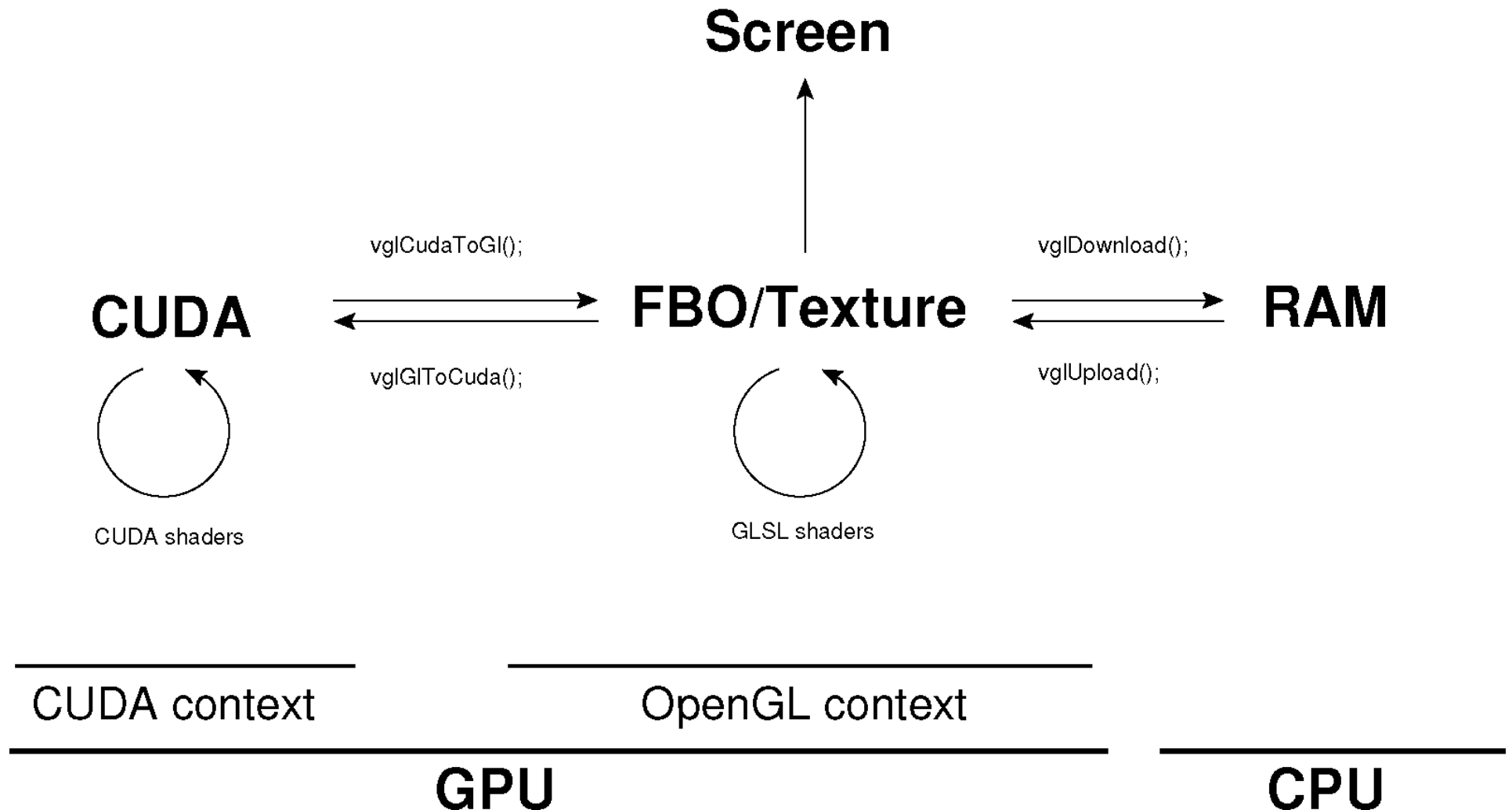
=



Usando OpenGL



Usando funções de alto nível



vglMipmap.frag

```
/** vglMipmap
    Get specified level of detail
 */

//(IN_TEX:  VglImage* src, OUT_FBO: VglImage*
    dst, float lod)

uniform sampler2D sampler0;
uniform float level; // lod

void main(void){
    gl_FragColor = texture2DLod(sampler0,
        gl_TexCoord[0].xy, level);
}
```

vglMipmap.frag

```
/** vglMipmap
    Get specified level of detail
 */

//(IN_TEX:  VglImage* src, OUT_FBO: VglImage*
    dst, float lod)

uniform sampler2D sampler0;
uniform float level; // lod

void main(void){
    gl_FragColor = texture2DLod(sampler0,
        gl_TexCoord[0].xy, level);
}
```

vglMipmap.frag

```
/** vglMipmap
    Get specified level of detail
 */

//(IN_TEX:  VglImage* src, OUT_FBO: VglImage*
    dst, float lod)

uniform sampler2D sampler0;
uniform float level; // lod

void main(void){
    gl_FragColor = texture2DLod(sampler0,
        gl_TexCoord[0].xy, level);
}
```

vglMipmap.frag

```
/** vglMipmap
    Get specified level of detail
 */

//(IN_TEX:  VglImage* src, OUT_FBO: VglImage*
    dst, float lod)

uniform sampler2D sampler0;
uniform float level; // lod

void main(void){
    gl_FragColor = texture2DLod(sampler0,
        gl_TexCoord[0].xy, level);
}
```

vglMipmap.frag

```
/** vglMipmap
    Get specified level of detail
 */

//(IN_TEX:  VglImage* src, OUT_FBO: VglImage*
    dst, float lod)

uniform sampler2D sampler0;
uniform float level; // lod

void main(void){
    gl_FragColor = texture2DLod(sampler0,
        gl_TexCoord[0].xy, level);
}
```

vglMipmap.frag

```
/** vglMipmap
    Get specified level of detail
 */

//(IN_TEX: VglImage* src, OUT_FBO: VglImage*
  dst, float lod)

uniform sampler2D sampler0;
uniform float level; // lod

void main(void){
    gl_FragColor = texture2DLod(sampler0,
        gl_TexCoord[0].xy, level);
}
```

vglMipmap.frag

```
/** vglMipmap
    Get specified level of detail
 */

//(IN_TEX:  VglImage* src, OUT_FBO: VglImage*
    dst, float lod)

uniform sampler2D sampler0;
uniform float level; // lod

void main(void){
    gl_FragColor = texture2DLod(sampler0,
        gl_TexCoord[0].xy, level);
}
```

vglMipmap.frag

```
/** vglMipmap
    Get specified level of detail
 */

//(IN_TEX:  VglImage* src, OUT_FBO: VglImage*
    dst, float lod)

uniform sampler2D sampler0;
uniform float level; // lod

void main(void){
    gl_FragColor = texture2DLod(sampler0,
        gl_TexCoord[0].xy, level);
}
```

vglMipmap.frag

```
/** vglMipmap
 * Get specified level of detail
 */
void vglMipmap(VglImage* src, VglImage* dst, float lod){
    glUniform1f(glGetUniformLocation(f, "level"), lod);

    vglCheckContext(src, VGL_GL_CONTEXT);

    GLint viewport[4];
    glGetIntegerv(GL_VIEWPORT, viewport);

    static GLuint f = 0;
    if (f == 0){
        fprintf(stdout, "FRAGMENT SHADER\n=====\n");
        f = vglShaderLoad(GL_FRAGMENT_SHADER, "FS/vglMipmap.frag");
        if (!f){
            fprintf(stderr, "%s: %s: Error loading fragment shader.\n", __FILE__,
                __FUNCTION__);
            exit(1);
        }
    }
    ERRCHECK()

    glUseProgram(f);

    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_2D, src->tex);
    glUniform1i(glGetUniformLocation(f, "sampler0"), 0);
    ERRCHECK()

    glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, dst->fbo);
    CHECK_FRAMEBUFFER_STATUS()
    ERRCHECK()

    ...

    glViewport(0, 0, 2*dst->width, 2*dst->height);

    glBegin(GL_QUADS);
    glTexCoord2f( 0.0, 0.0);
    glVertex3f (-1.0, -1.0, 0.0); //Left Up
    glTexCoord2f( 1.0, 0.0);
    glVertex3f ( 0.0, -1.0, 0.0); //Right Up
    glTexCoord2f( 1.0, 1.0);
    glVertex3f ( 0.0, 0.0, 0.0); //Right Bottom
    glTexCoord2f( 0.0, 1.0);
    glVertex3f (-1.0, 0.0, 0.0); //Left Bottom
    glEnd();

    glUseProgram(0);

    glViewport(viewport[0], viewport[1], viewport[2], viewport[3]);

    if (dst->has_mipmap){
        glBindTexture(GL_TEXTURE_2D, dst->tex);
        glGenerateMipmapEXT(GL_TEXTURE_2D);
    }
    glActiveTexture(GL_TEXTURE0);

    vglSetContext(dst, VGL_GL_CONTEXT);

}
```

vglCudaInvert.kernel

```
/** vglCudaInvert
    Inverts image stored in cuda context.
 */
// <<<input->height,384>>> (IN_PBO: VglImage* input,
    OUT_PBO: VglImage* output)
// (input->cudaPtr, output->cudaPtr, input->width,
    input->height, input->nChannels)
template<typename T>
__global__ void global_Invert(T* input, T* output, int
    w, int h, int nChannels){
    int offset = blockIdx.x * nChannels * w;
    T* array_in = input + offset;
    T* array_out = output + offset;
    for (int j = threadIdx.x; j < nChannels * w; j +=
        blockDim.x){
        array_out[j] = -array_in[j];
    }
}
```

vglCudaInvert.kernel

```
/** vglCudaInvert
    Inverts image stored in cuda context.
 */
// <<<input->height,384>>> (IN_PBO: VglImage* input,
    OUT_PBO: VglImage* output)
// (input->cudaPtr, output->cudaPtr, input->width,
    input->height, input->nChannels)
template<typename T>
__global__ void global_Invert(T* input, T* output, int
    w, int h, int nChannels){
    int offset = blockIdx.x * nChannels * w;
    T* array_in = input + offset;
    T* array_out = output + offset;
    for (int j = threadIdx.x; j < nChannels * w; j +=
        blockDim.x){
        array_out[j] = -array_in[j];
    }
}
```

vglCudaInvert.kernel

```
/** vglCudaInvert
    Inverts image stored in cuda context.
 */
// <<<input->height,384>>> (IN_PBO: VglImage* input,
    OUT_PBO: VglImage* output)
// (input->cudaPtr, output->cudaPtr, input->width,
    input->height, input->nChannels)
template<typename T>
__global__ void global_Invert(T* input, T* output, int
    w, int h, int nChannels){
    int offset = blockIdx.x * nChannels * w;
    T* array_in = input + offset;
    T* array_out = output + offset;
    for (int j = threadIdx.x; j < nChannels * w; j +=
        blockDim.x){
        array_out[j] = -array_in[j];
    }
}
```

vglCudaInvert.kernel

```
/** vglCudaInvert
    Inverts image stored in cuda context.
 */
// <<<input->height,384>>> (IN_PBO: VglImage* input,
    OUT_PBO: VglImage* output)
// (input->cudaPtr, output->cudaPtr, input->width,
    input->height, input->nChannels)
template<typename T>
__global__ void global_Invert(T* input, T* output, int
    w, int h, int nChannels){
    int offset = blockIdx.x * nChannels * w;
    T* array_in = input + offset;
    T* array_out = output + offset;
    for (int j = threadIdx.x; j < nChannels * w; j +=
        blockDim.x){
        array_out[j] = -array_in[j];
    }
}
```

vglCudaInvert.kernel

```
/** vglCudaInvert
    Inverts image stored in cuda context.
 */
// <<<input->height,384>>> (IN_PBO: VglImage* input,
    OUT_PBO: VglImage* output)
// (input->cudaPtr, output->cudaPtr, input->width,
    input->height, input->nChannels)
template<typename T>
__global__ void global_Invert(T* input, T* output, int
    w, int h, int nChannels){
    int offset = blockIdx.x * nChannels * w;
    T* array_in = input + offset;
    T* array_out = output + offset;
    for (int j = threadIdx.x; j < nChannels * w; j +=
        blockDim.x){
        array_out[j] = -array_in[j];
    }
}
```

vglCudaInvert.kernel

```
/** vglCudaInvert
    Inverts image stored in cuda context.
 */
// <<<input->height,384>>> (IN_PBO: VglImage* input,
    OUT_PBO: VglImage* output)
// (input->cudaPtr, output->cudaPtr, input->width,
    input->height, input->nChannels)
template<typename T>
__global__ void global_Invert(T* input, T* output, int
    w, int h, int nChannels){
    int offset = blockIdx.x * nChannels * w;
    T* array_in = input + offset;
    T* array_out = output + offset;
    for (int j = threadIdx.x; j < nChannels * w; j +=
        blockDim.x){
        array_out[j] = -array_in[j];
    }
}
```

vglCudaInvert.kernel

```
/** vglCudaInvert
    Inverts image stored in cuda context.
 */
// <<<input->height,384>>> (IN_PBO: VglImage* input,
    OUT_PBO: VglImage* output)
// (input->cudaPtr, output->cudaPtr, input->width,
    input->height, input->nChannels)
template<typename T>
__global__ void global_Invert(T* input, T* output, int
    w, int h, int nChannels){
    int offset = blockIdx.x * nChannels * w;
    T* array_in = input + offset;
    T* array_out = output + offset;
    for (int j = threadIdx.x; j < nChannels * w; j +=
        blockDim.x){
        array_out[j] = -array_in[j];
    }
}
```

```

/** vglCudaInvert
 * Inverts image stored in cuda context.
 */
void vglCudaInvert(VglImage* input, VglImage* output){
    if (!input){
        fprintf(stderr, "vglCudaInvert: Error: input parameter is null in file '%s' in line %i.\n", __FILE__, __LINE__);
        exit(1);
    }
    vglCheckContext(input, VGL_CUDA_CONTEXT);
    if (!input->cudaPtr){
        fprintf(stderr, "vglCudaInvert: Error: input->cudaPtr is null in file '%s' in line %i.\n", __FILE__, __LINE__);
        exit(1);
    }
    if (!output){
        fprintf(stderr, "vglCudaInvert: Error: output parameter is null in file '%s' in line %i.\n", __FILE__, __LINE__);
        exit(1);
    }
    vglCheckContextForOutput(output, VGL_CUDA_CONTEXT);
    if (!output->cudaPtr){
        fprintf(stderr, "vglCudaInvert: Error: output->cudaPtr is null in file '%s' in line %i.\n", __FILE__, __LINE__);
        exit(1);
    }
    switch (input->depth){
        case (IPL_DEPTH_8U):
            global_Invert<<<input->height,384>>>((unsigned char* )input->cudaPtr, (unsigned char* )output->cudaPtr,
            input->width, input->height, input->nChannels);
            break;
        default:
            fprintf(stderr, "vglCudaInvert: Error: unsupported img->depth = %d in file '%s' in line %i.\n",
                input->depth, __FILE__, __LINE__);
            exit(1);
    }
    vglSetContext(output, VGL_CUDA_CONTEXT);
}

```

Hello.cpp

```
#include <visiongl.h>

int main(int argc, char** argv){
    VglImage* img_in      =
        vglLoadImage("/home/ddantas/images/lena.tif",
            CV_LOAD_IMAGE_GRAYSCALE);
    VglImage* img_out     = vglCreateImage(img_in);

    cvNamedWindow("image in",          CV_WINDOW_AUTOSIZE);
    cvNamedWindow("image out",         CV_WINDOW_AUTOSIZE);

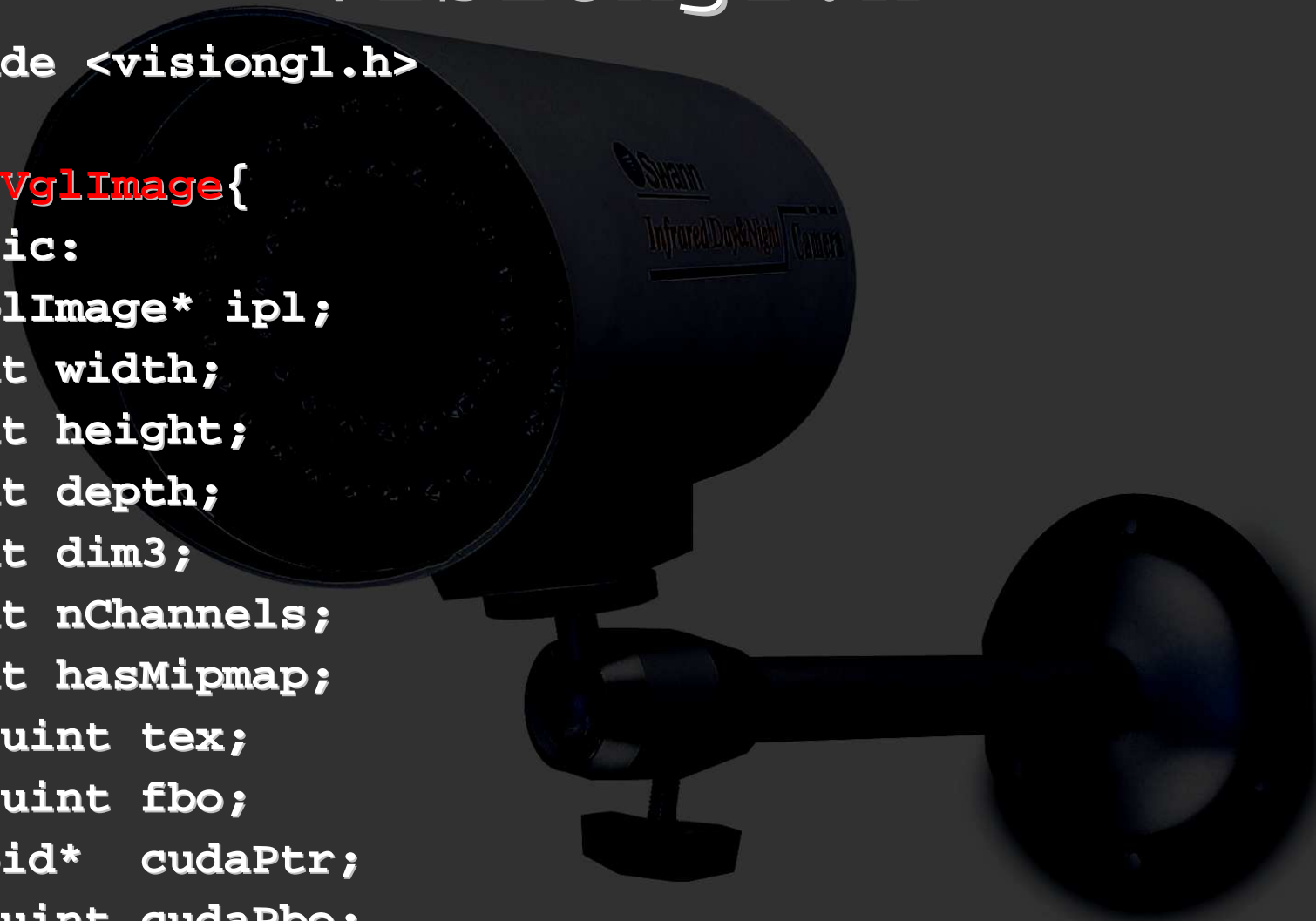
    vglMipMap(img_in, img_out, 2);
    vglDownload(img_out);

    cvShowImage(winname_out,          img_out->ipl);
    return 0;
}
```

visiongl.h

```
#include <visiongl.h>
```

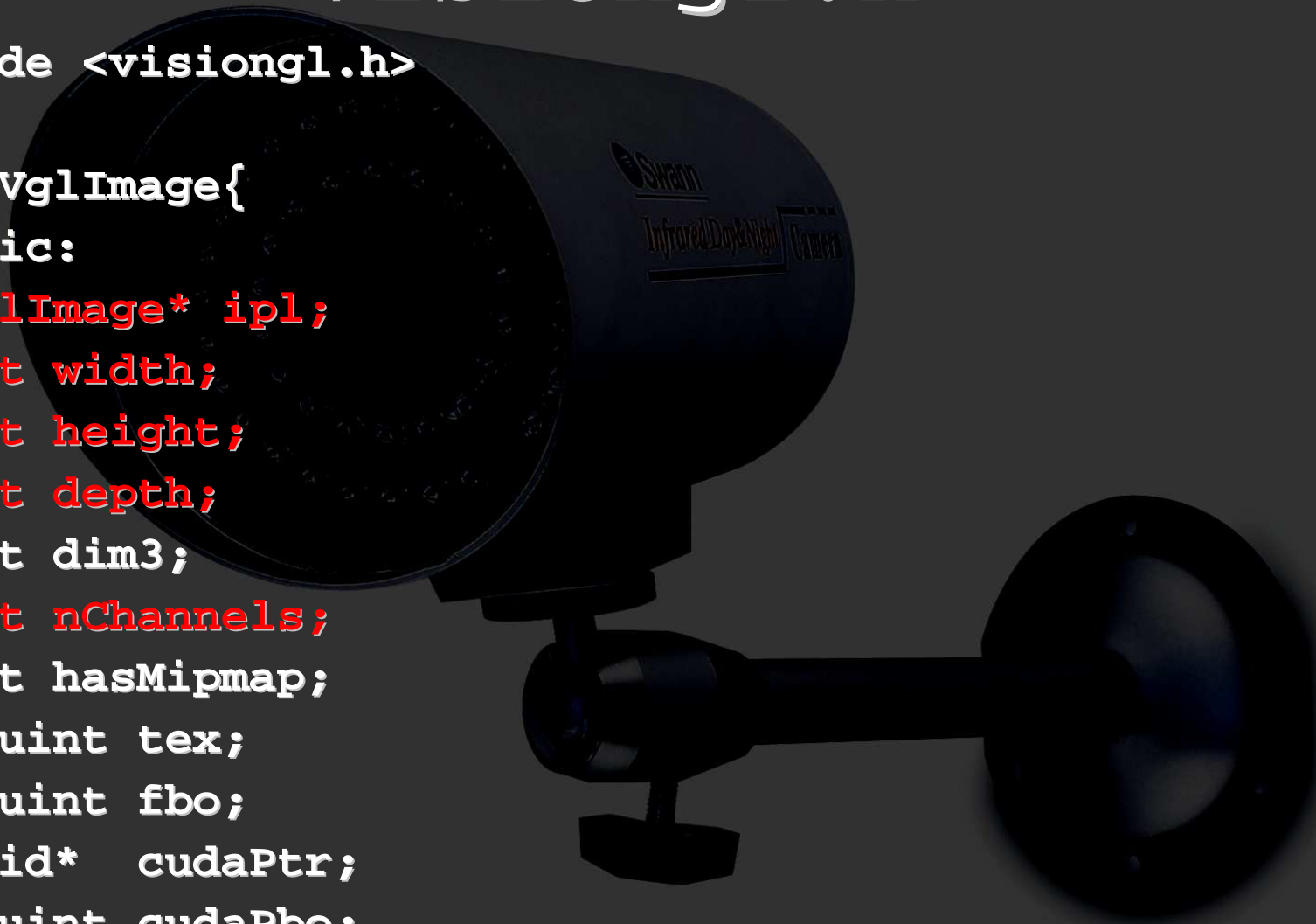
```
Class VglImage{  
    public:  
        IplImage* ipl;  
        int width;  
        int height;  
        int depth;  
        int dim3;  
        int nChannels;  
        int hasMipmap;  
        Gluint tex;  
        Gluint fbo;  
        void* cudaPtr;  
        Gluint cudaPbo;  
        int inContext;  
}
```



visiongl.h

```
#include <visiongl.h>
```

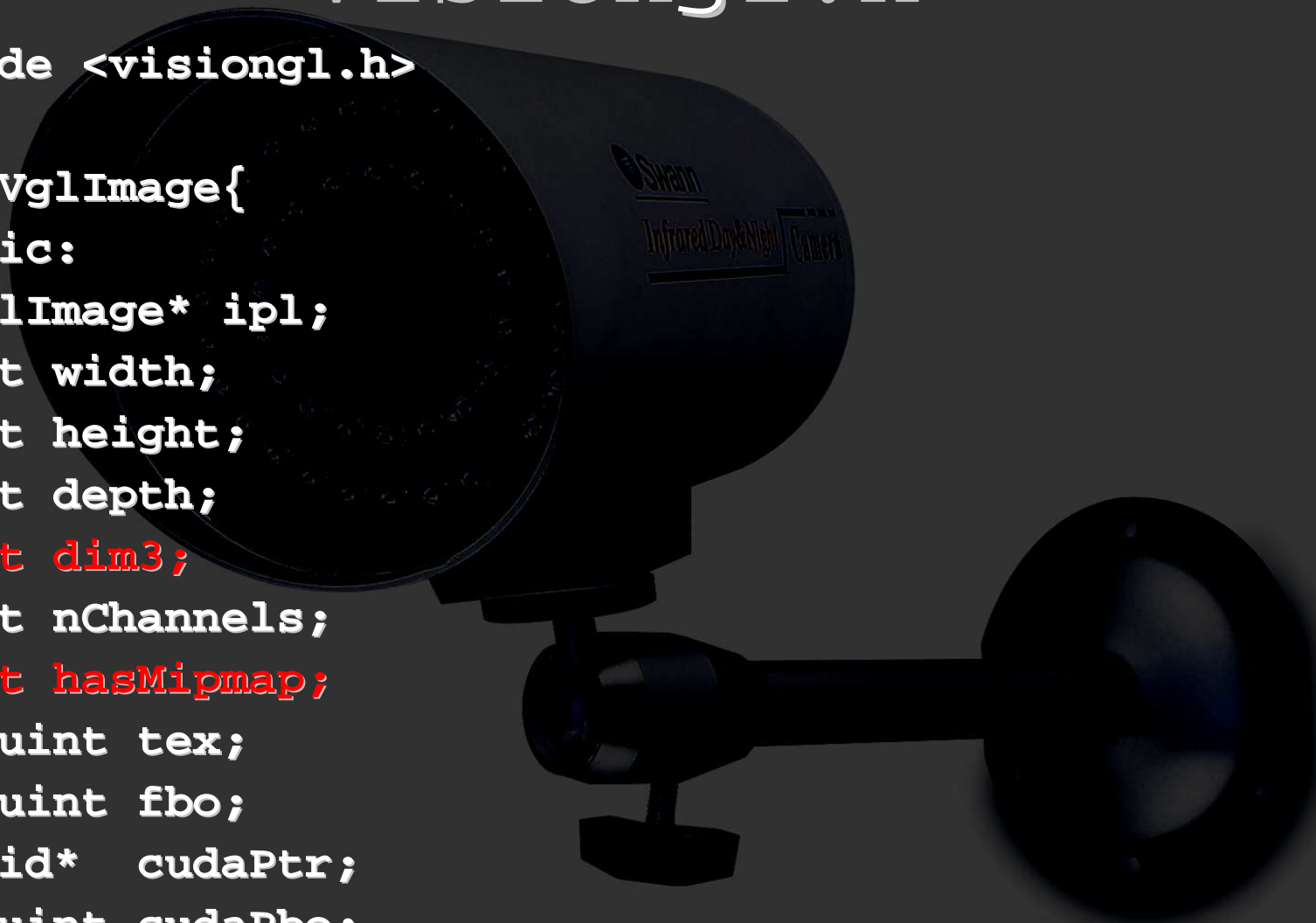
```
Class VglImage{  
    public:  
        IplImage* ipl;  
        int width;  
        int height;  
        int depth;  
        int dim3;  
        int nChannels;  
        int hasMipmap;  
        Gluint tex;  
        Gluint fbo;  
        void* cudaPtr;  
        Gluint cudaPbo;  
        int inContext;  
}
```



visiongl.h

```
#include <visiongl.h>
```

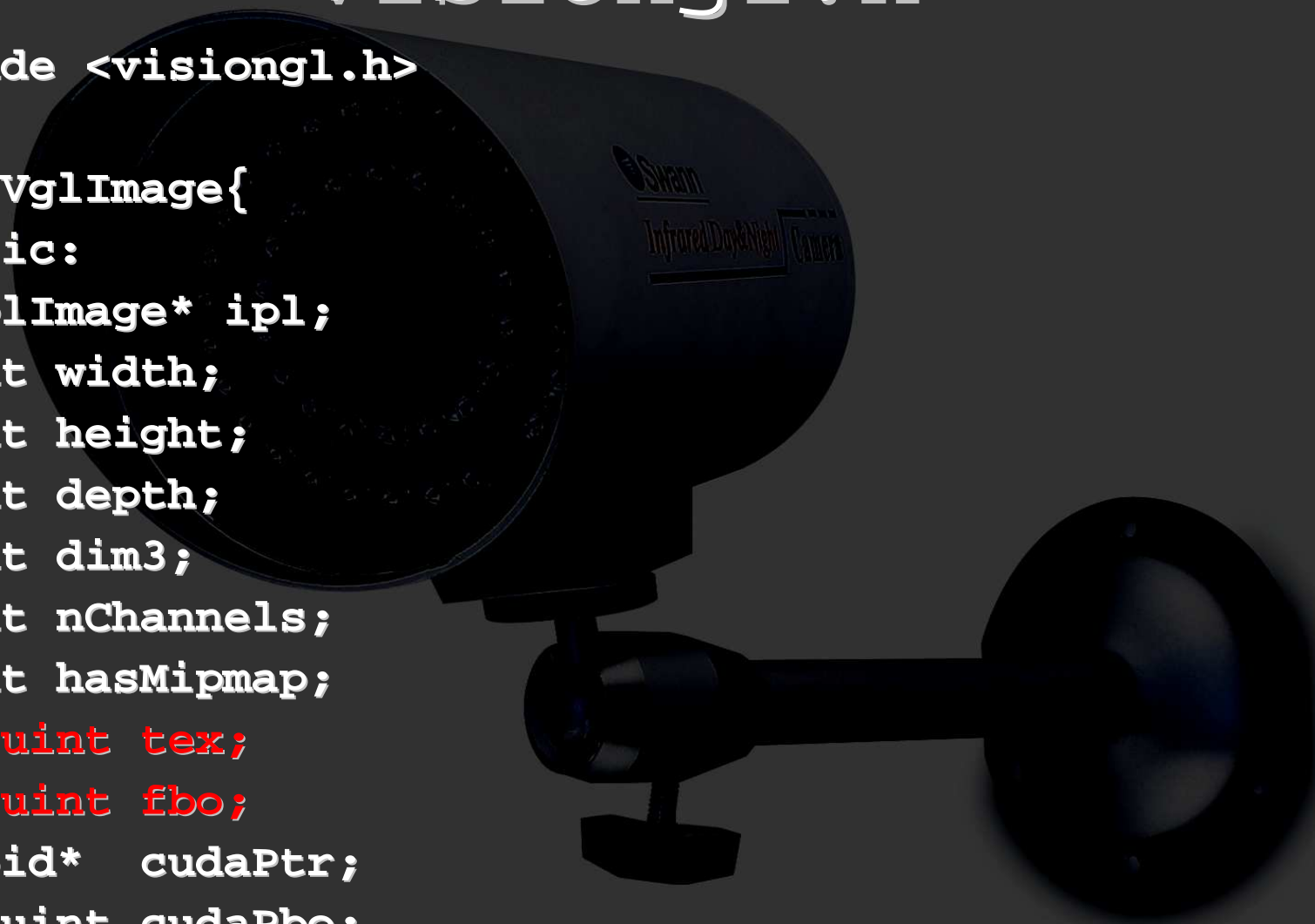
```
Class VglImage{  
    public:  
        IplImage* ipl;  
        int width;  
        int height;  
        int depth;  
        int dim3;  
        int nChannels;  
        int hasMipmap;  
        Gluint tex;  
        Gluint fbo;  
        void* cudaPtr;  
        Gluint cudaPbo;  
        int inContext;  
}
```



visiongl.h

```
#include <visiongl.h>
```

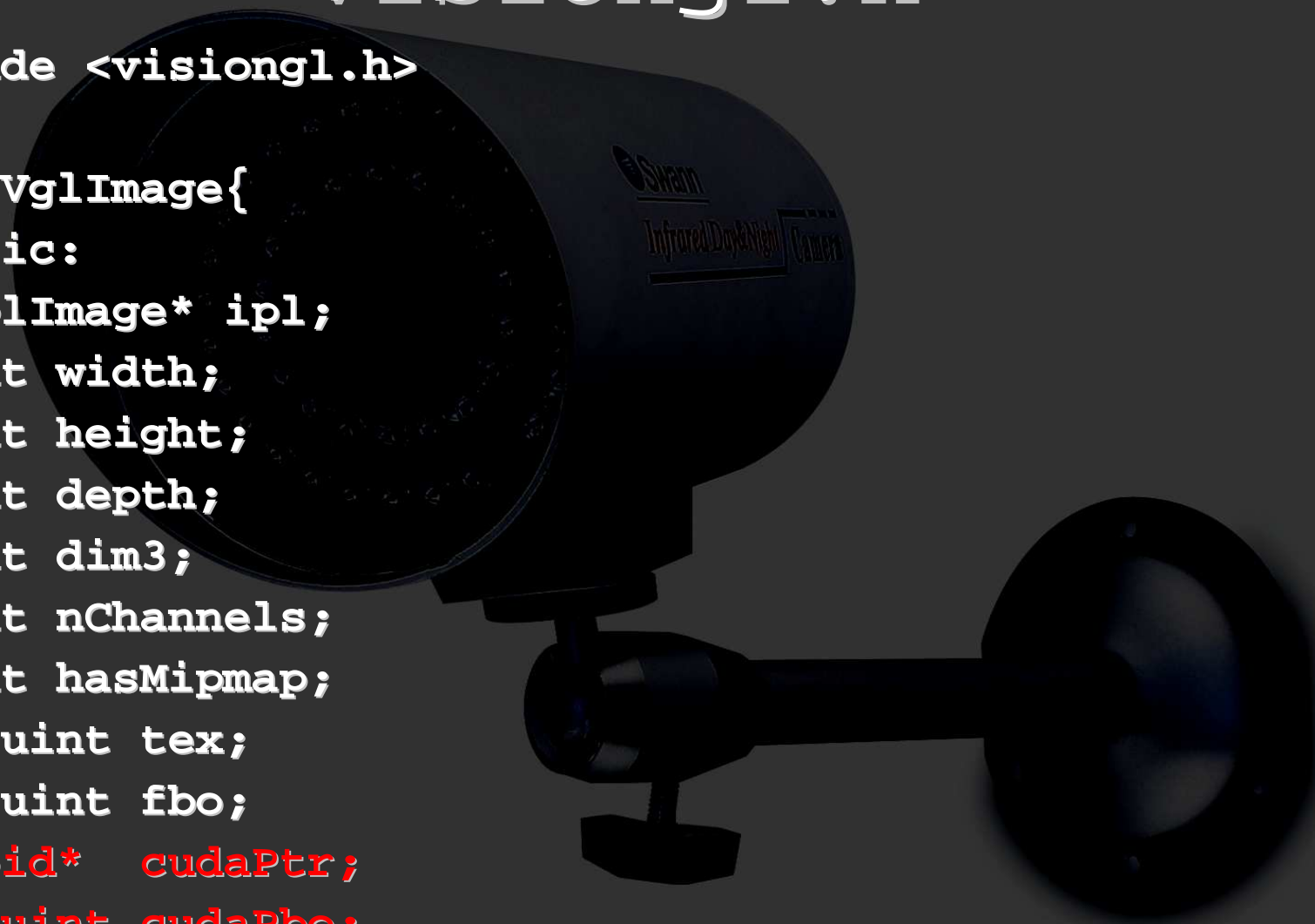
```
Class VglImage{  
    public:  
        IplImage* ipl;  
        int width;  
        int height;  
        int depth;  
        int dim3;  
        int nChannels;  
        int hasMipmap;  
        Gluint tex;  
        Gluint fbo;  
        void* cudaPtr;  
        Gluint cudaPbo;  
        int inContext;  
}
```



visiongl.h

```
#include <visiongl.h>
```

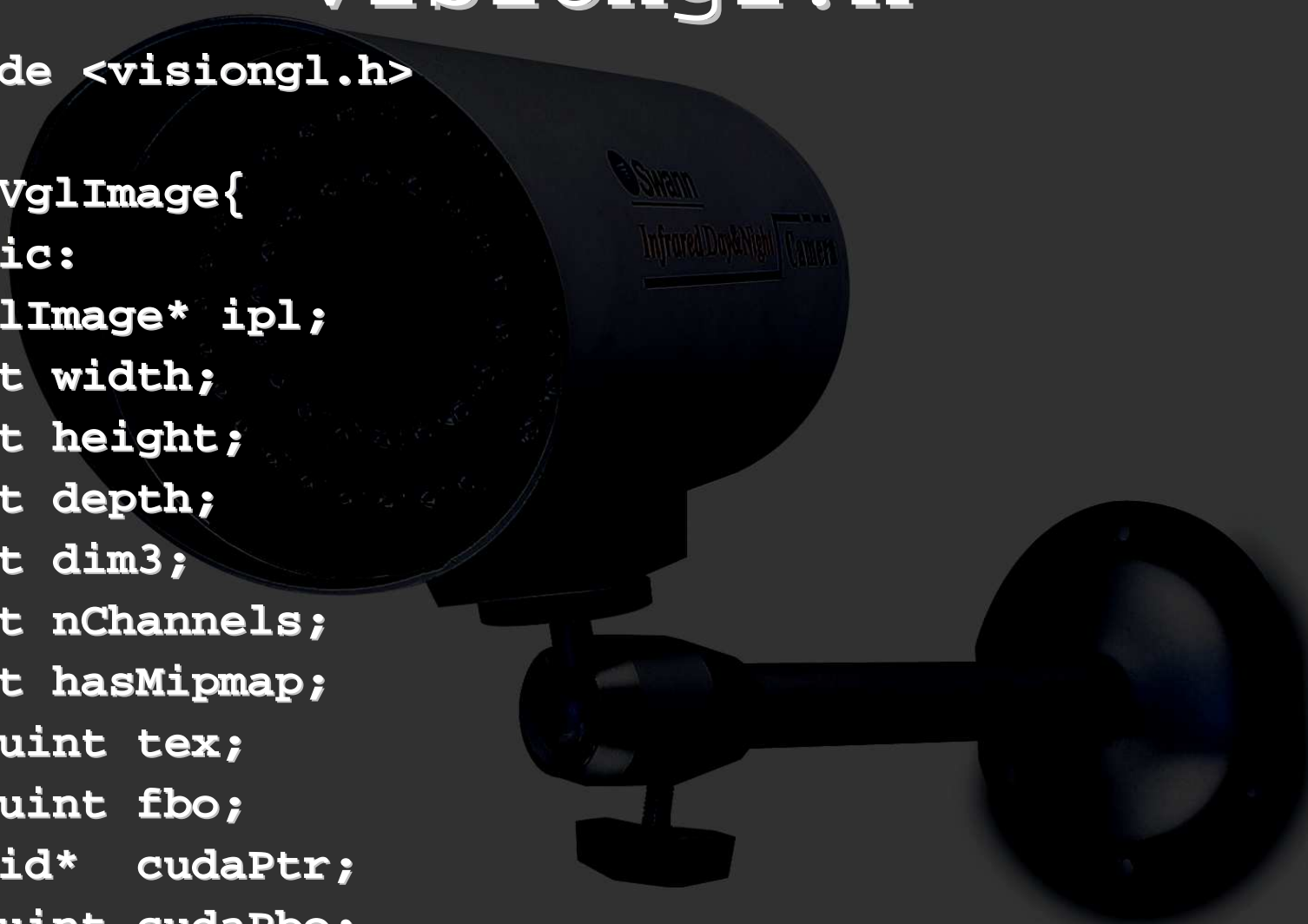
```
Class VglImage{  
    public:  
        IplImage* ipl;  
        int width;  
        int height;  
        int depth;  
        int dim3;  
        int nChannels;  
        int hasMipmap;  
        Gluint tex;  
        Gluint fbo;  
        void* cudaPtr;  
        Gluint cudaPbo;  
        int inContext;  
}
```



visiongl.h

```
#include <visiongl.h>
```

```
Class VglImage{  
    public:  
        IplImage* ipl;  
        int width;  
        int height;  
        int depth;  
        int dim3;  
        int nChannels;  
        int hasMipmap;  
        Gluint tex;  
        Gluint fbo;  
        void* cudaPtr;  
        Gluint cudaPbo;  
        int inContext;  
}
```



Tempos de processamento

	OpenCV	GPUCV	VGL
Erosão 3x3	38.4	1.6	0.8
Erosão 5x5	63.4	3.3	2.1
RGB→XYZ	11.9	0.8	0.2
RGB→HSV	21.0	1.0	0.3
Threshold	2.2	0.7	0.2
Copy	3.1	1.3	0.2
Subtraction	7.8	0.9	0.3
CPU→GPU		10.3	5.1
GPU→CPU		5.3	8.5

The background is a dark, blurry image. In the center, there is a bright red light source with a greenish-yellow core. The overall scene is dimly lit, with some faint blue and white highlights on the left and right sides.

Obrigado!

Perguntas?