GLSL Programming

Nicolas Holzschuch

GLSL programming

- C-like language structure:

```
int i, j;
i = 2;
j = 0;
j += i;
```

- Functions, loops, branches...
- *Reading* your first GSL shader is easy

Differences with C

- New data types: vec, mat...
- Must write *simple* code
- Data input/output
- Compiling & linking

New data types

- `vec4`: a 4-component floating-point array
  - also `vec2`, `vec3`, `ivec4`...
- `mat4`: a 4x4 floating-point matrix
  - also `mat2`, `mat3`...
- Standard operations on vec/mat

Standard operations

- `v = u + w`
  - does the right thing with vec/mat
- `v = M*w`
  - Matrix-vector multiplication, matrix-matrix multiplication...
- Other operators (see reference):
  - `length(v), dot(v,w), cross(v,w)...`

Standard math functions

- `sin, cos, tan…`
- `asin, acos, atan…`
- `pow, exp, log, sqrt, inversesqrt`
- `abs, sign, frac, floor, min, max,`
- `step, smoothstep`
- Use them! (don’t recode them)

Components & swizzle

- Access to vector components:
  - `u.x` equivalent to `u[0]`
- Operations on several components:
  - `u.xyz = v.xyz/v.w` (projection)
- Reordering:
  - `u = u.wzyx; w = u.xxyy;`

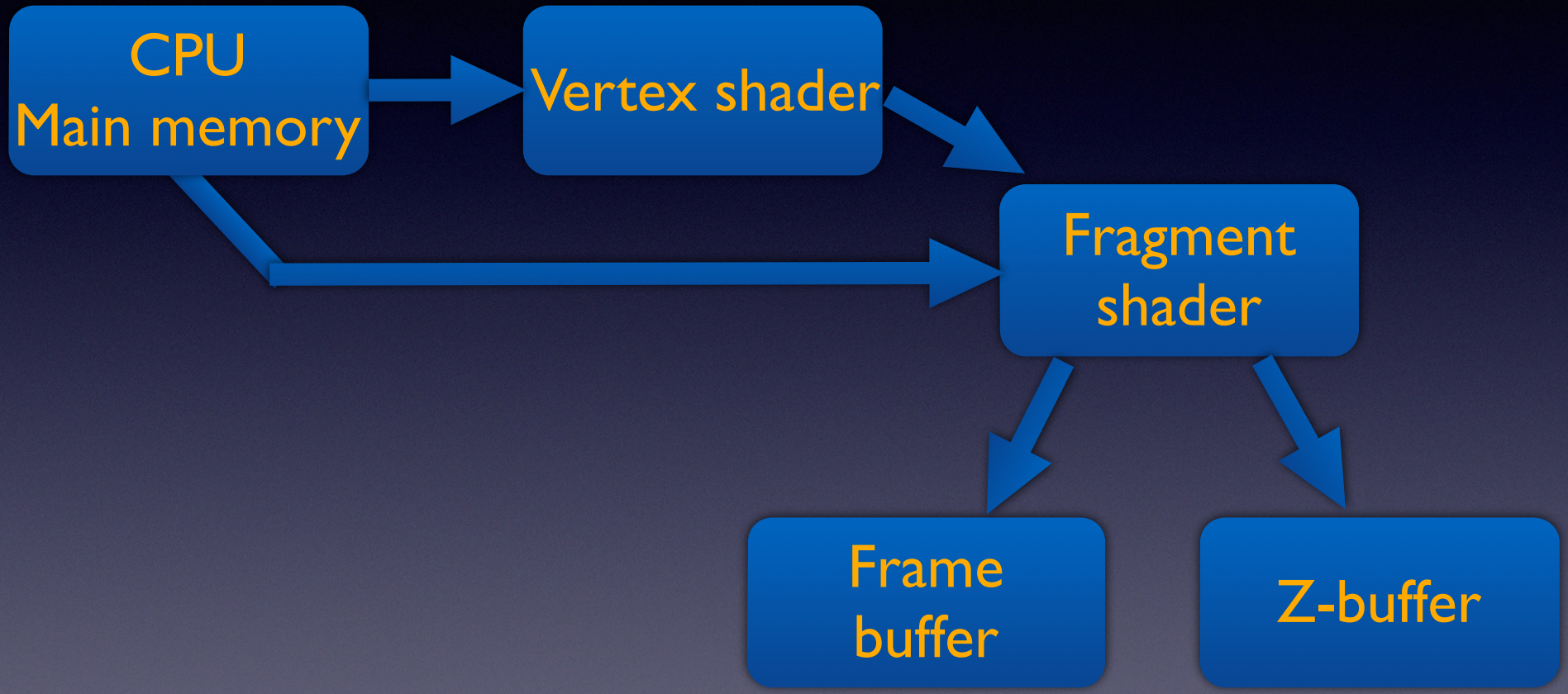
Components & swizzle

- *Very* useful for Image Synthesis
- Several sets of letters:
  - `u.xyzw / u.rgba / u.stpq`
- Can use any of them
- Makes code easier to read

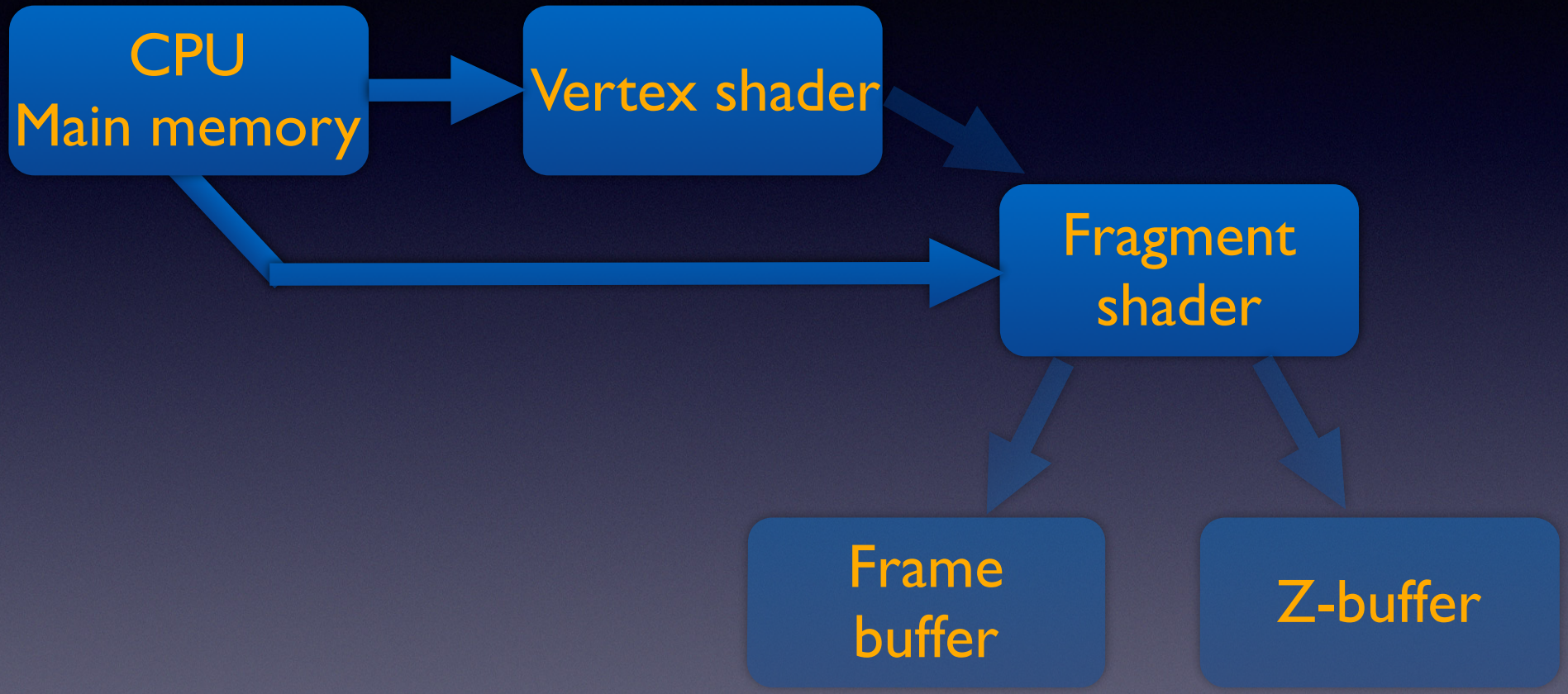
Input & Output

- The basis for any program

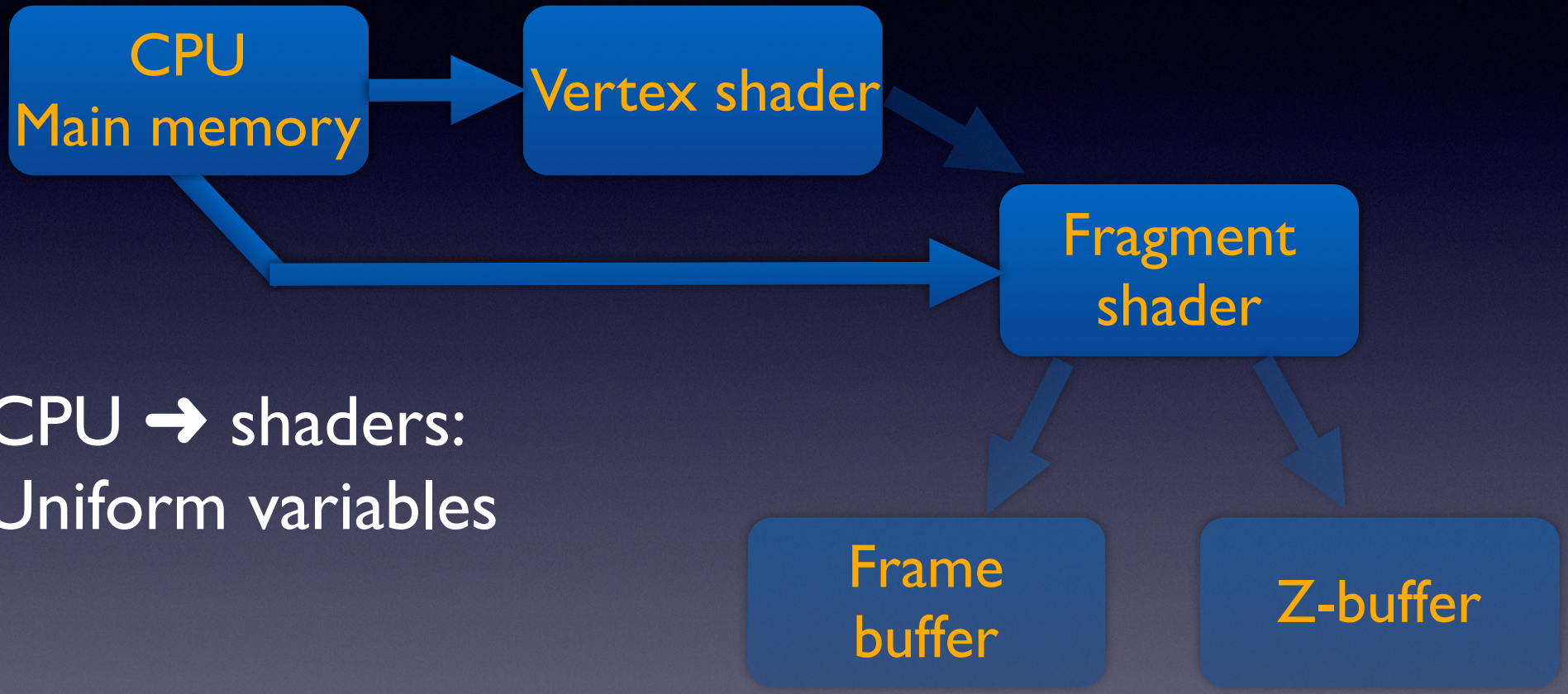
Input & Output



Input & Output

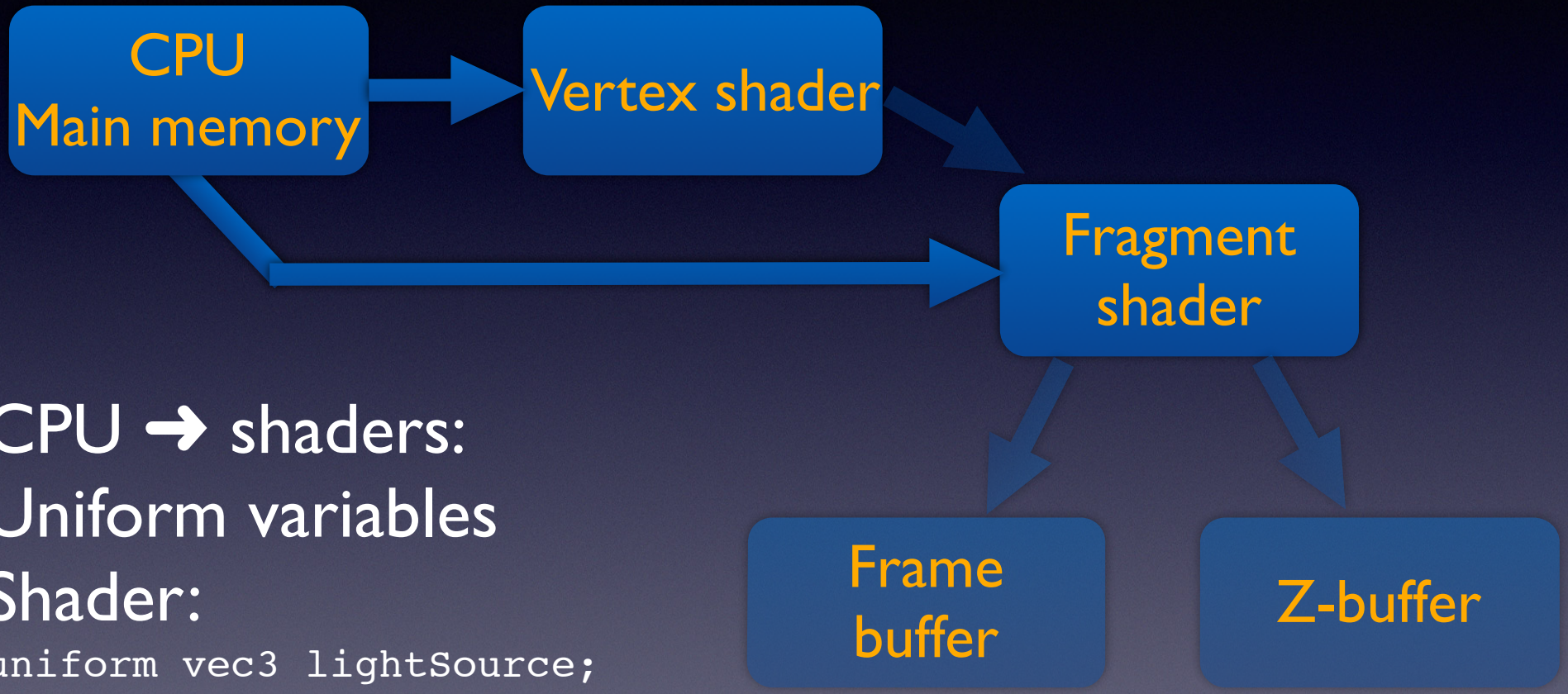


Input & Output



CPU ➜ shaders:
Uniform variables

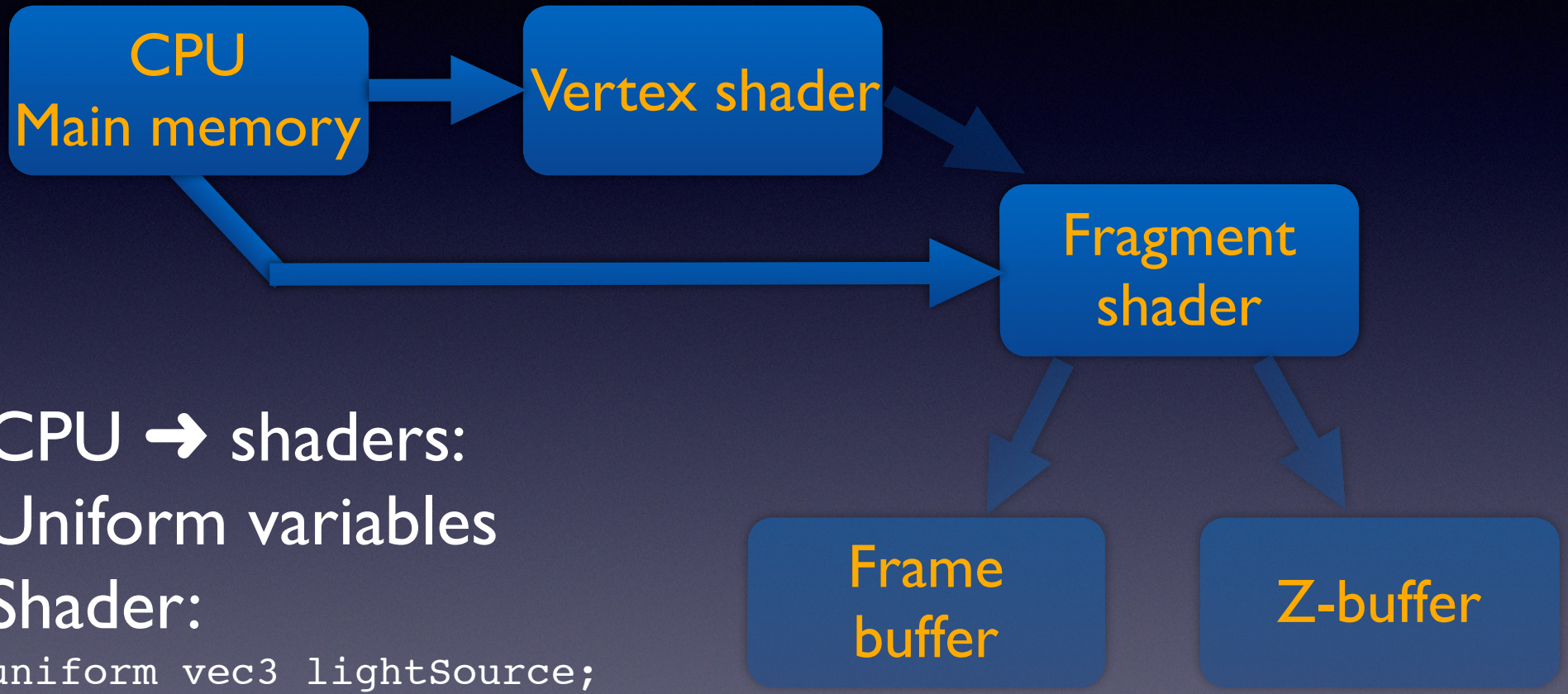
Input & Output



CPU ➜ shaders:
Uniform variables
Shader:

```
uniform vec3 lightSource;
```

Input & Output



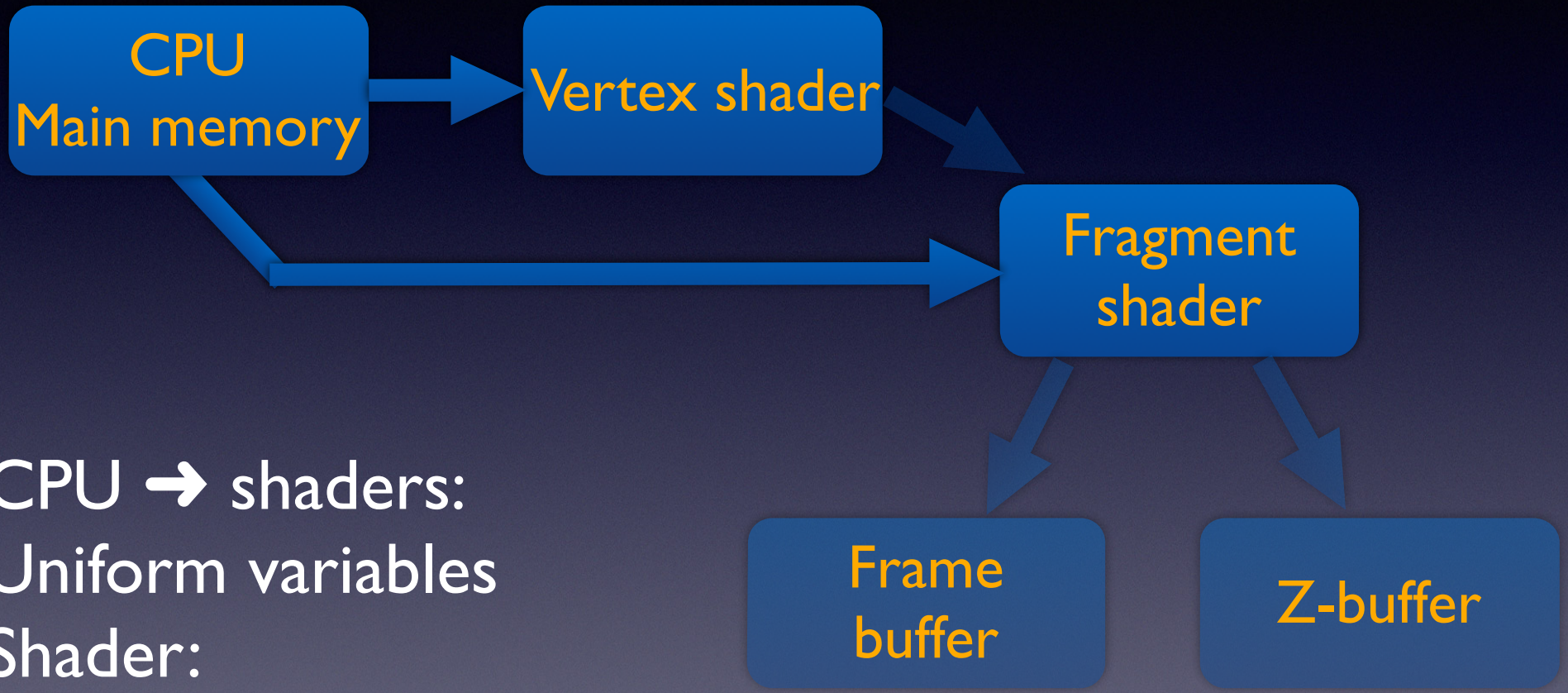
CPU ➜ shaders:
Uniform variables
Shader:

```
uniform vec3 lightSource;
```

Program:

```
loc=glGetUniformLocation(shader, "lightSource");
glUniform3f(loc, x, y, z);
```

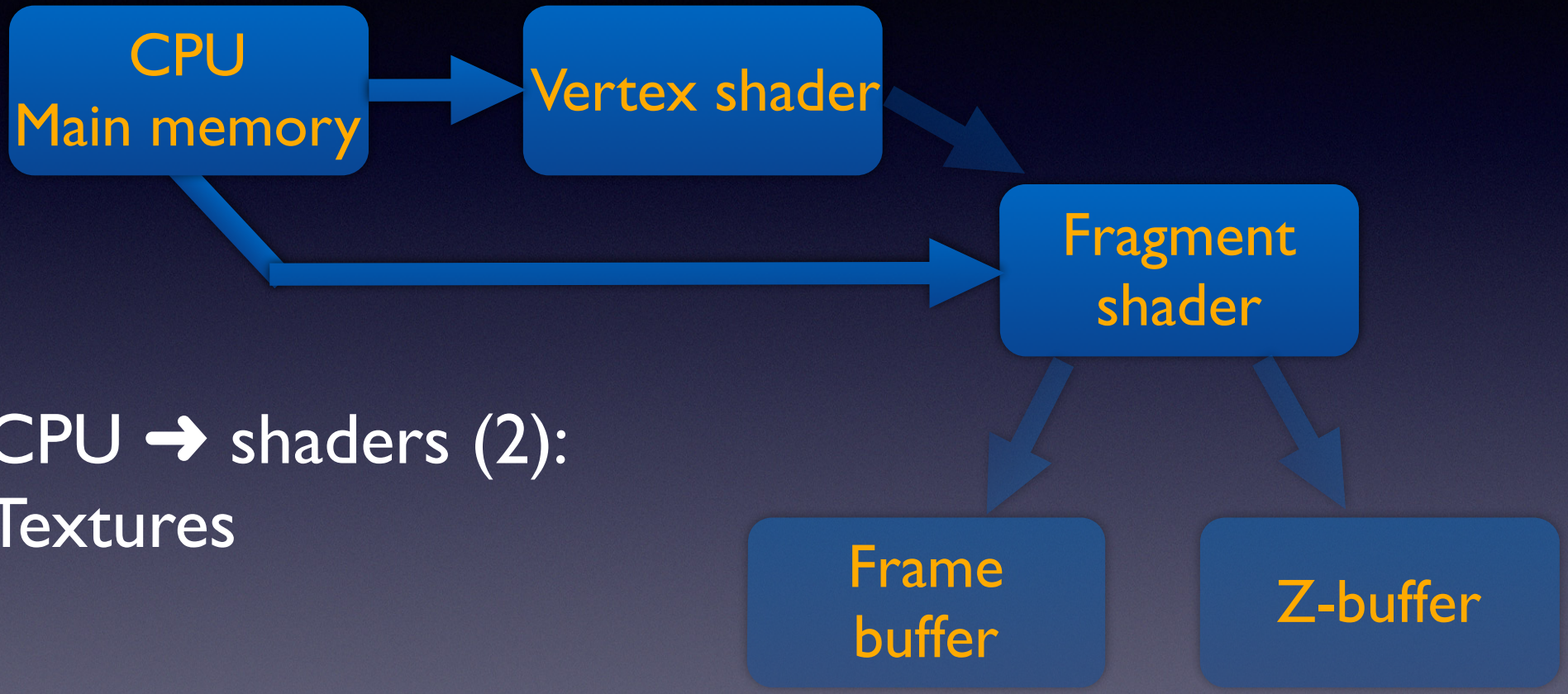
Input & Output



CPU ➜ shaders:
Uniform variables
Shader:

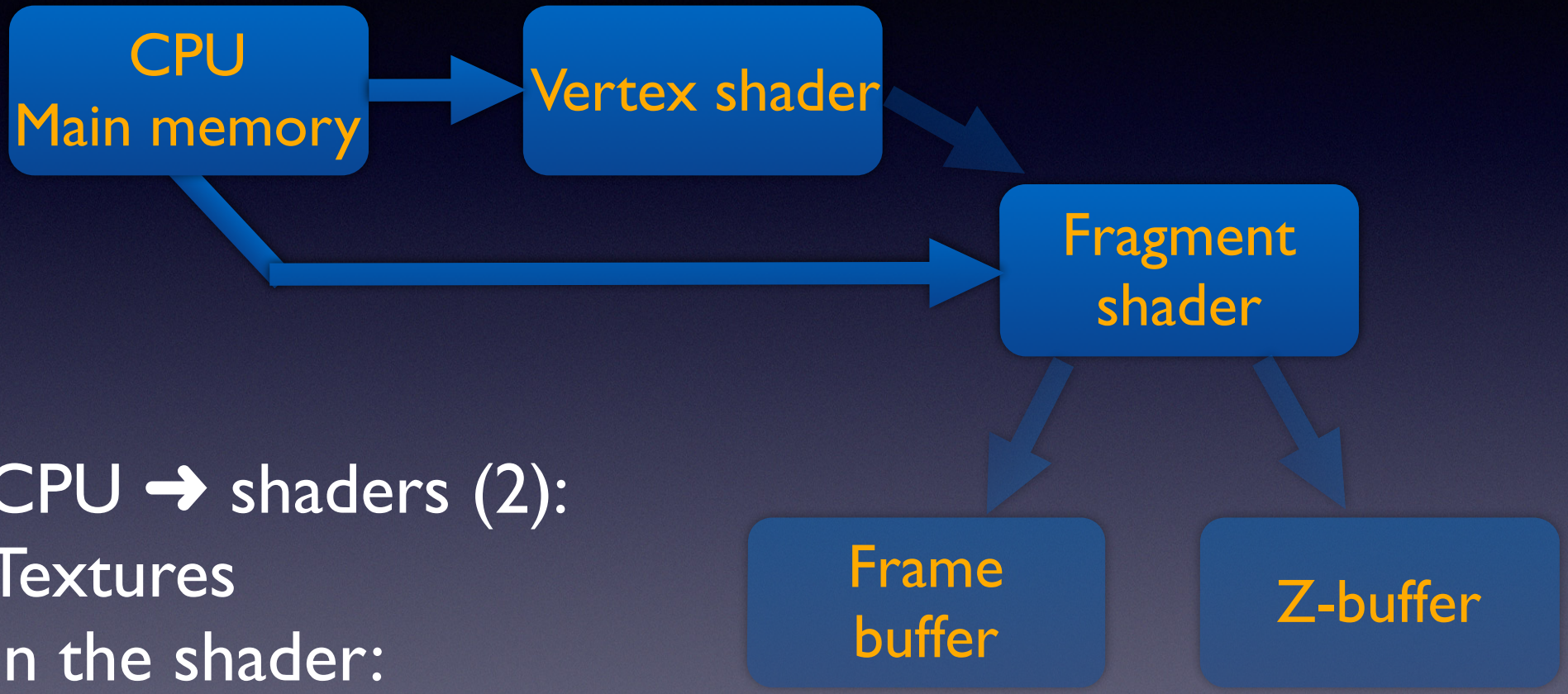
```
uniform vec3 lightSource;
```

Input & Output



CPU ➜ shaders (2):
Textures

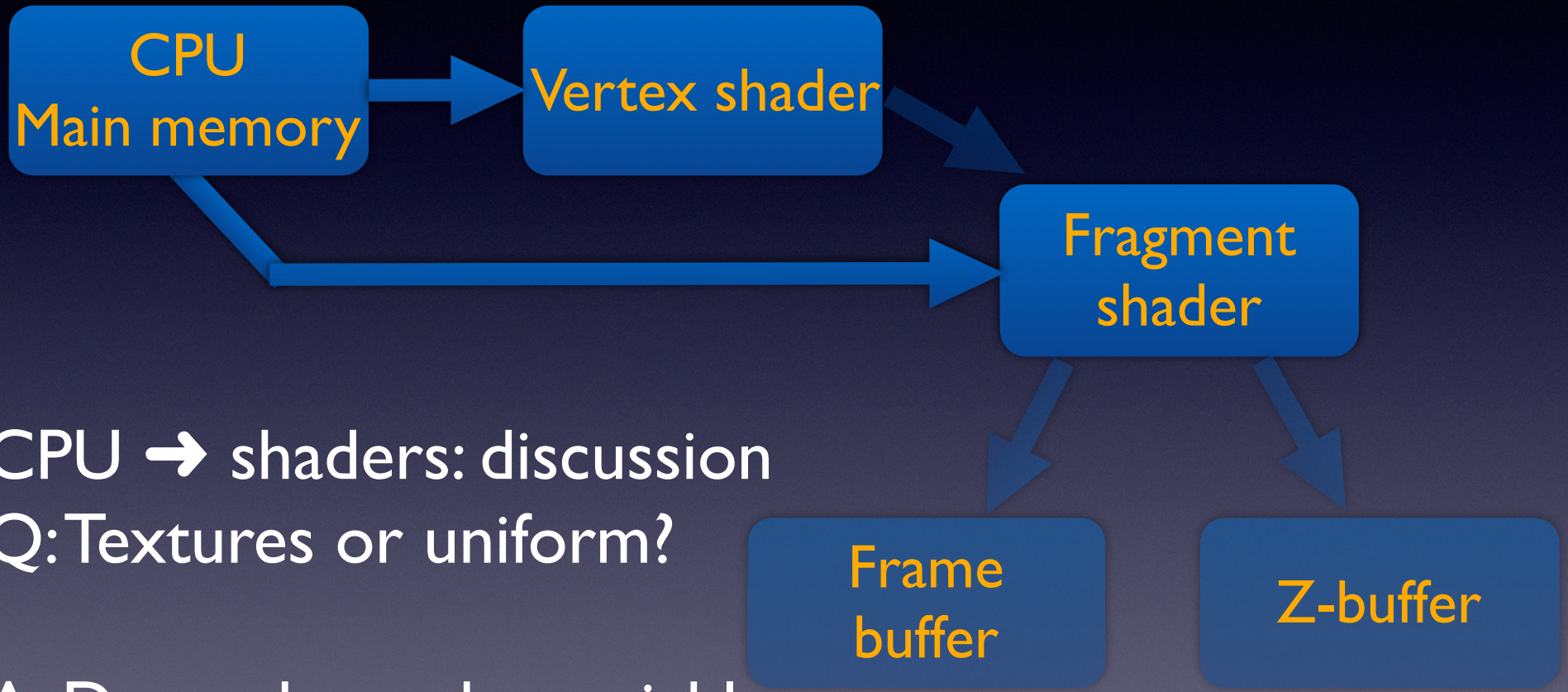
Input & Output



CPU ➜ shaders (2):
Textures
In the shader:

```
uniform sampler2D MyTex;
```

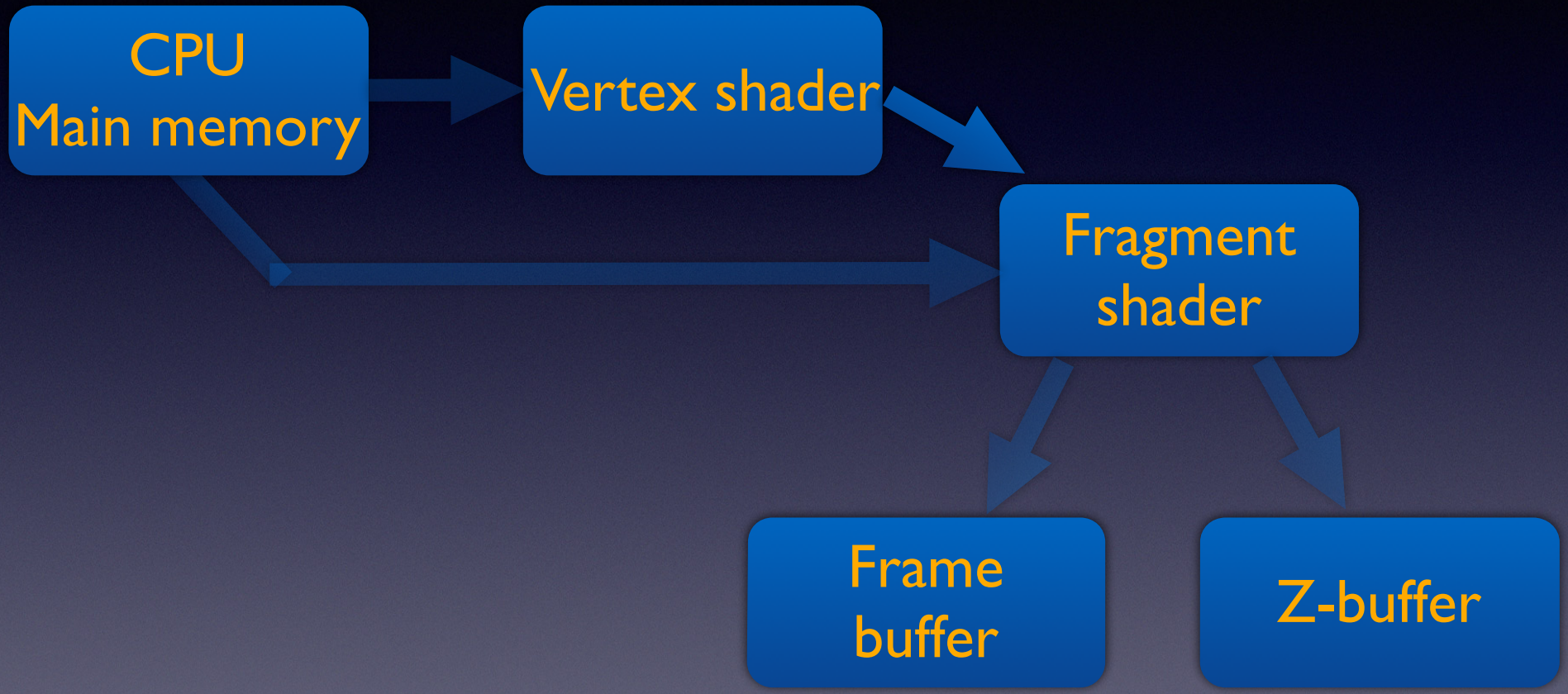
Input & Output



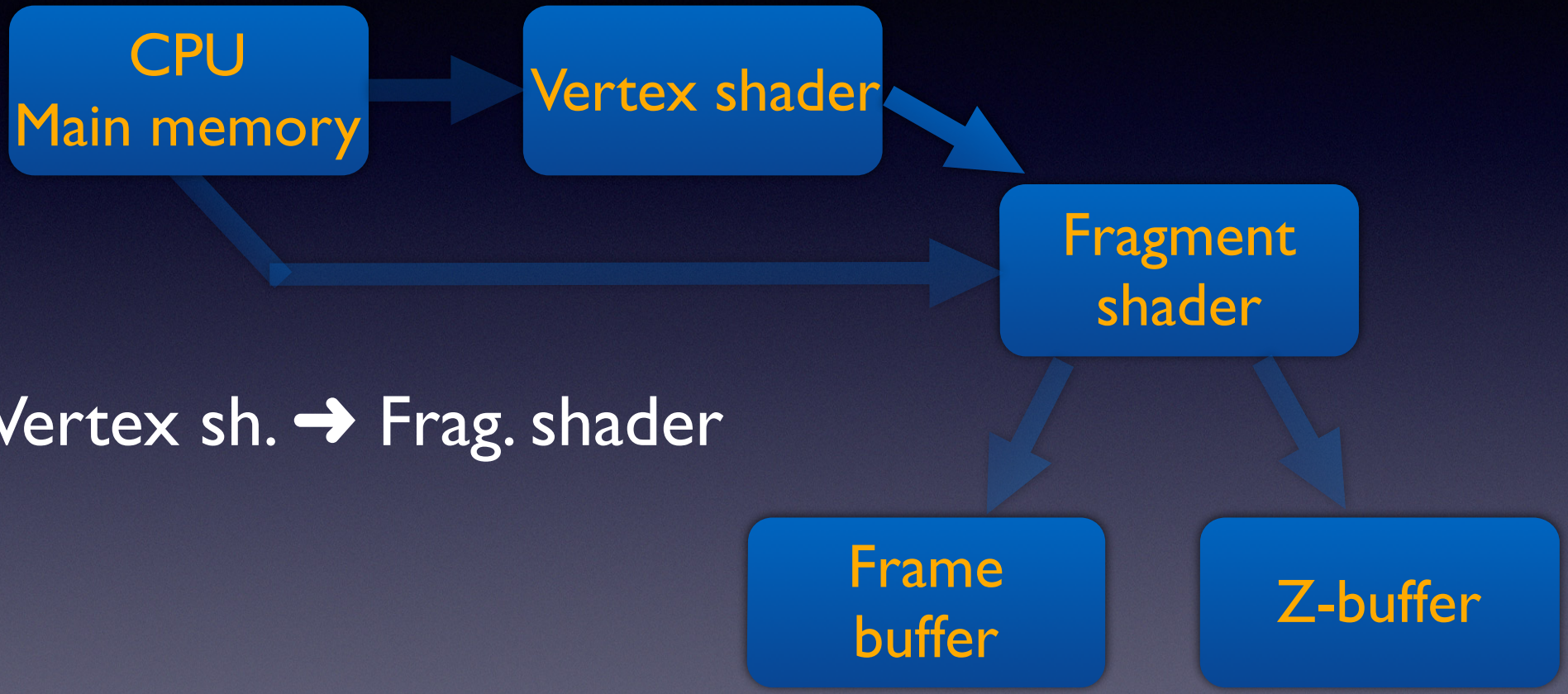
CPU ➜ shaders: discussion
Q: Textures or uniform?

A: Depends on the variable
Constant (for every vertex): uniform
Depends on position: texture

Input & Output

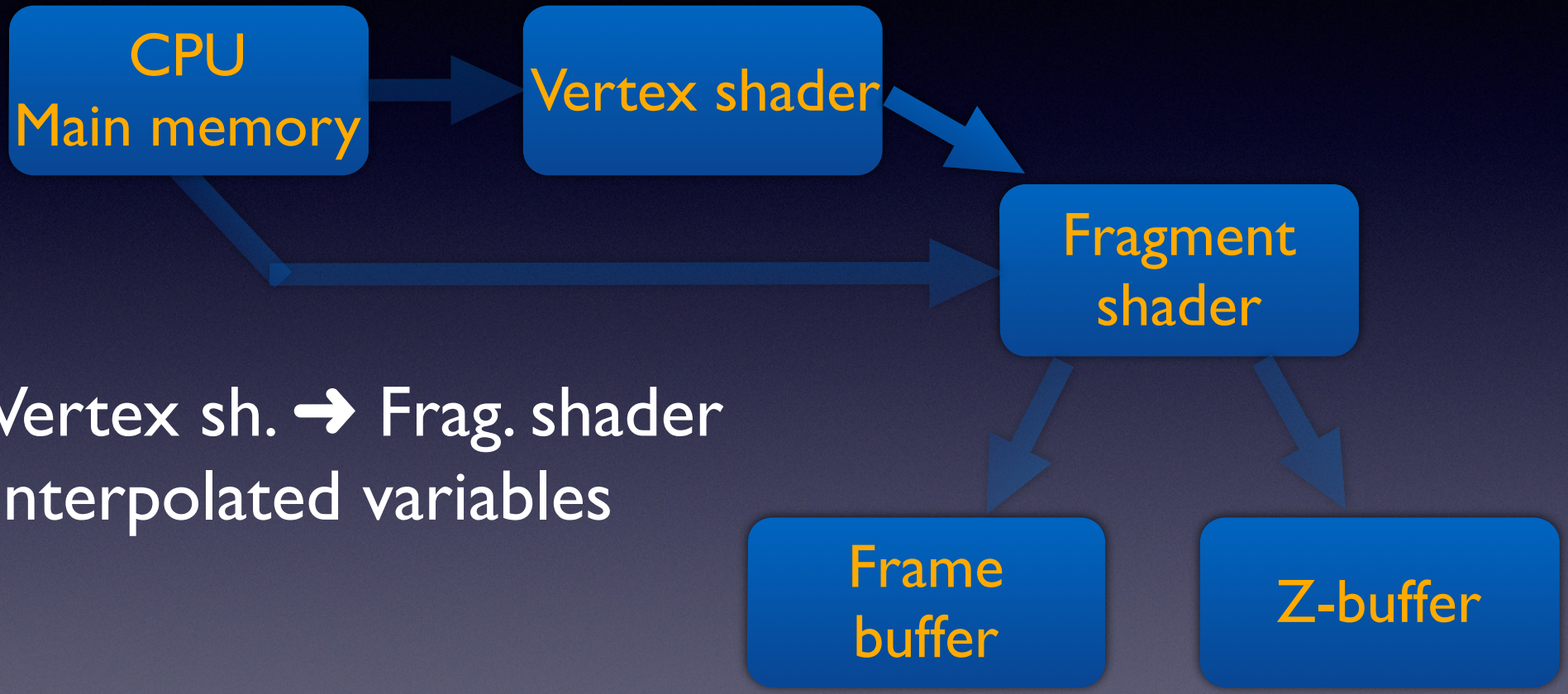


Input & Output



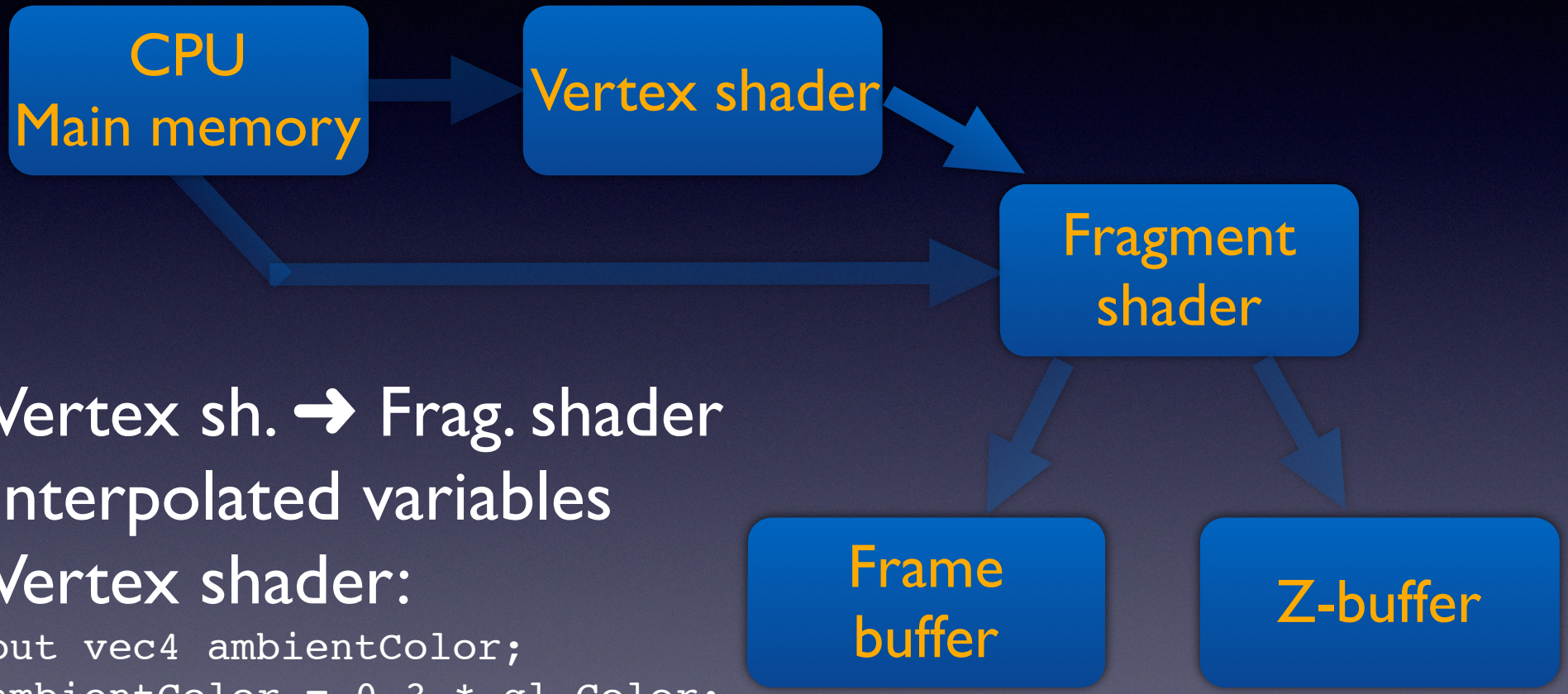
Vertex sh. ➜ Frag. shader

Input & Output



Vertex sh. ➜ Frag. shader
Interpolated variables

Input & Output



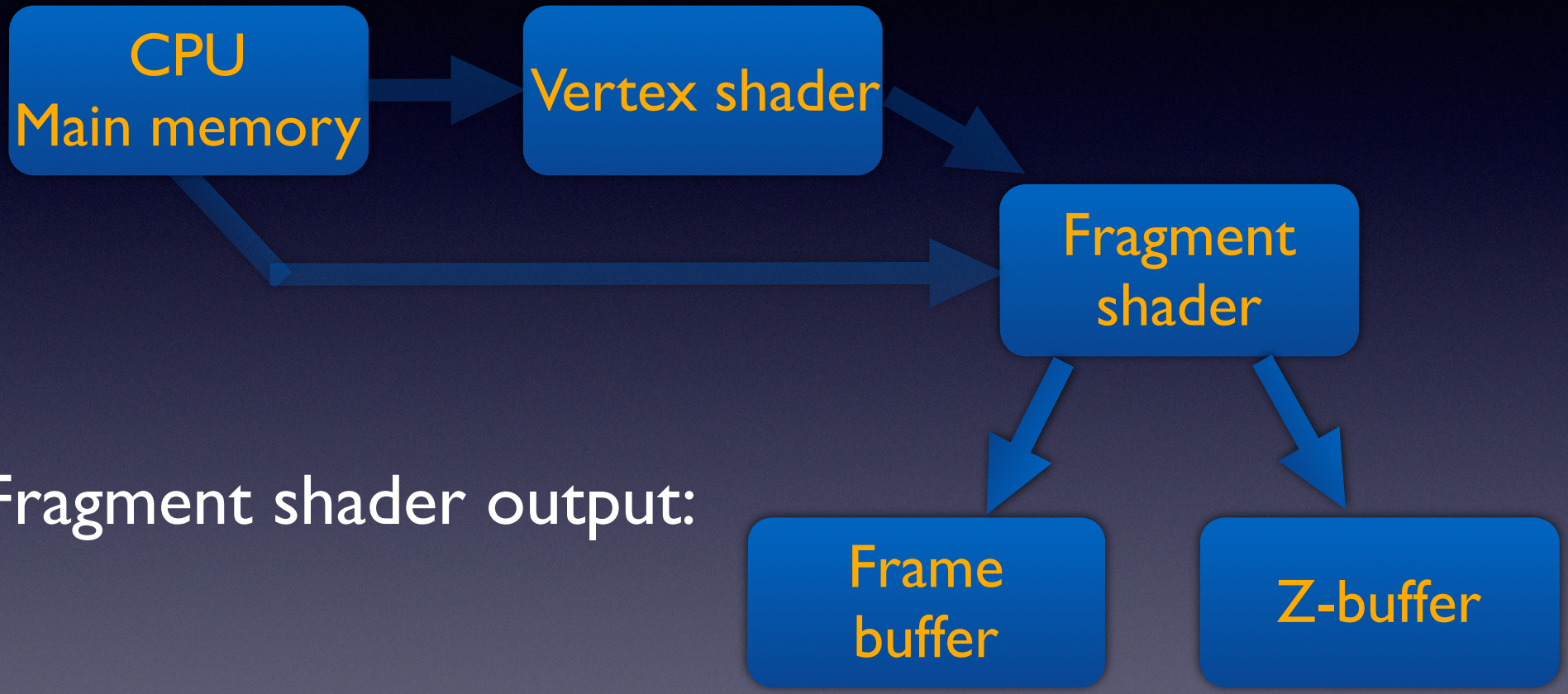
Vertex sh. ➜ Frag. shader
Interpolated variables
Vertex shader:

```
out vec4 ambientColor;
ambientColor = 0.3 * gl_Color;
```

Fragment shader:

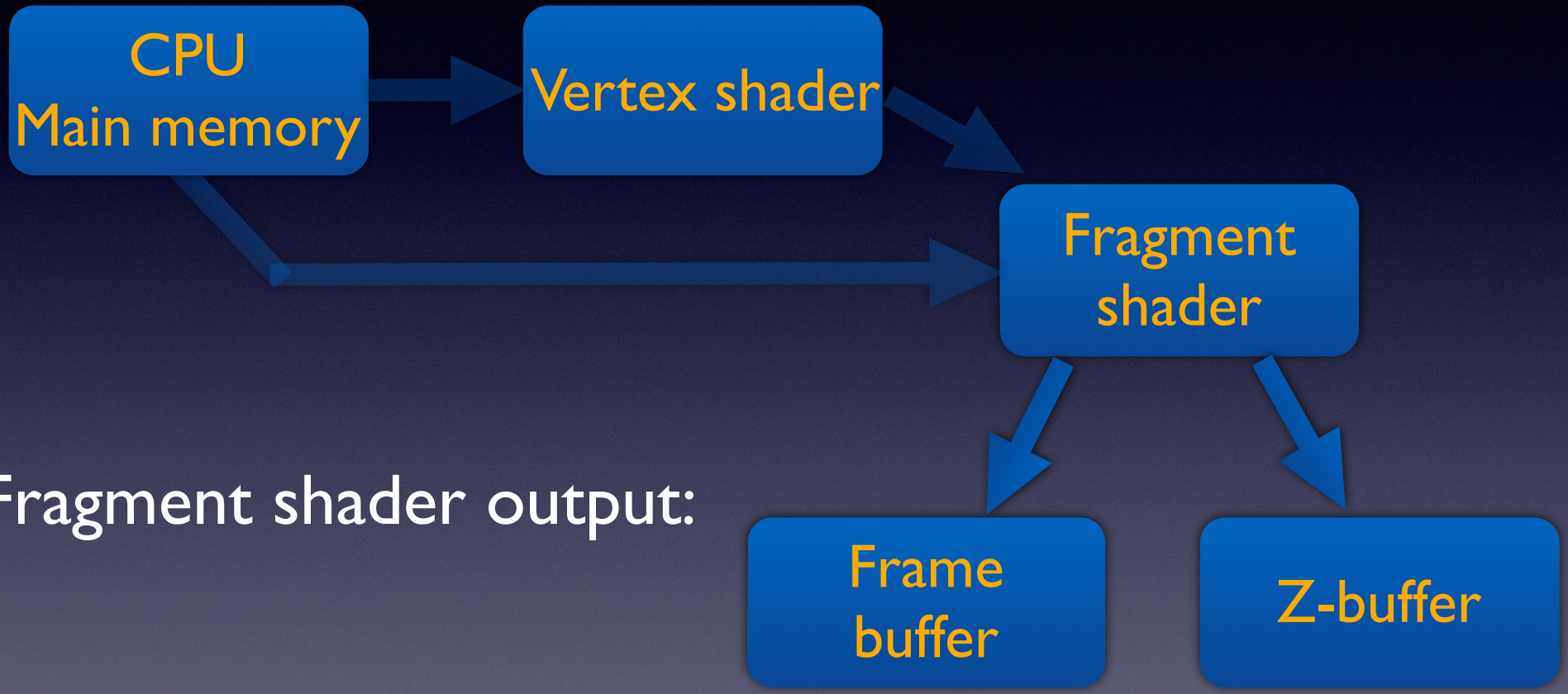
```
in vec4 ambientColor;
```

Input & Output



Fragment shader output:

Input & Output



Fragment shader output:

```
fragColor = ...; //required
fragDepth = ...; //optional
```

Compiling & linking

- 3 steps:
  - load and compile vertex shader
  - load and compile fragment shader
  - link shader
- OpenGL functions:
  - `glShaderSource, glCreateShader, glCompileShader, glLinkProgram`

Compiling & linking

- 3 steps:
  - load and compile vertex shader
  - load and compile fragment shader
  - link shader

Compiling & linking

- Check the result of the compilation

```
if (newProg != NULL) ...
```

- Get the errors! (`glGetShaderInfoLog`)
- Read them, they're your only information

One fun bug

- Compiler designed for efficiency
- Removes everything useless
- …including useless variables
- can be quite aggressive in that
- …and complains of non-existent variable
- also, C++ program tests if variable exists…

That's it!

- These are the basics
- You have a working program to start with
- Edit it, understand it, improve it
- Start with simple programs

OpenGL4 and Qt5

- OpenGL 4: removes many legacy options
  - more efficient, but different
- Qt5: great integration with OpenGL4
  - encapsulates everything useful

Loading a shader

```
QOpenGLShaderProgram* program = new
QOpenGLShaderProgram(this);
program->addShaderFromSourceFile
(QOpenGLShader::Vertex, vertexShaderPath);
program->addShaderFromSourceFile
(QOpenGLShader::Fragment,fragmentShaderPath);
program->link();
program->bind();
```

Loading a shader (2)

```
QOpenGLShaderProgram* program = new
QOpenGLShaderProgram(this);
if (!program) qWarning() << "Failed to allocate the
shader";
bool result = program->addShaderFromSourceFile
(QOpenGLShader::Vertex, vertexShaderPath);
if ( !result ) qWarning() << program->log();
result = program->addShaderFromSourceFile
(QOpenGLShader::Fragment, fragmentShaderPath);
if ( !result ) qWarning() << program->log();
result = program->link();
if ( !result ) qWarning() << program->log();
program->bind();
return program;
```

Passing variables

```
m_program->bind();
m_program->setUniformValue
("lightPosition",lightPosition);
```

Encoding the scene

- Before (Open GL 3 and below): face-based

```
glBegin(GL_POLYGON)
   glVertex3f(x1,y1,z1);
   glVertex3f(x2,y2,z2);
   glVertex3f(x3,y3,z3);
   glVertex3f(x4,y4,z4);
glEnd();
```

Encoding the scene

- OpenGL4: Vertex Array Object
  - object-based
- Single structure for each object
- Vertex, color, normals, tex coords in arrays
- VAO = set of arrays (vertex, color…)

Vertex Array Objects: creation

```
QOpenGLVertexArrayObject m_vao;

QOpenGLBuffer m_vertexBuffer;
QOpenGLBuffer m_indexBuffer;
QOpenGLBuffer m_normalBuffer;
QOpenGLBuffer m_colorBuffer;
QOpenGLBuffer m_texcoordBuffer;
```

Vertex array objects: filling

```
m_vao.bind();
m_vertexBuffer.setUsagePattern(QOpenGLBuffer::StaticDra
w);
m_vertexBuffer.bind();
m_vertexBuffer.allocate(&(modelMesh->vertices.front()),
modelMesh->vertices.size()*sizeof(trimesh::point));
```

Pointer

Size (bytes)

(same for index, color, normals…)

Vertex array objects: drawing

```
m_program->bind();
m_vao.bind();
glDrawElements(GL_TRIANGLES,
3 * modelMesh->faces.size(),
GL_UNSIGNED_INT, 0 );
m_vao.release();
m_program->release();
```

Primitive

Nb index

starting index

Notes

- Only the VAO is drawn
  - Internal buffers used only at creation time
- What if I only want to draw the geometry?
  - driver-dependent
  - = doesn't always work
  - drivers expect vertices+normals

Textures

```
texture = new QOpenGLTexture
(QImage(textureName));

texture->setWrapMode(QOpenGLTexture::Repeat);

texture->setMinificationFilter
(QOpenGLTexture::LinearMipMapLinear);

texture->setMagnificationFilter
(QOpenGLTexture::Linear);

texture->bind(0);

m_program->setUniformValue("colorTexture", 0);
```

Texture unit number

Frame Buffer Objects

- For offscreen rendering
  - shadow maps, deferred shading
  - multi-target rendering

Frame Buffer Objects

```
shadowMap = new QOpenGLFramebufferObject(w,h);
shadowMap->bind();
```

From here, all render events go to FBO

```
shadowMap->release();
m_program->setUniformValue("shadowMap", shadowMap->texture());
```

And we make it a texture for main shader

Frame Buffer Objects

```
QOpenGLFramebufferObjectFormat sFormat;
sFormat.setAttachment(QOpenGLFramebufferObject::Depth);
sFormat.setTextureTarget(GL_TEXTURE_2D);
sFormat.setInternalTextureFormat (GL_RGBA32F_ARB);
shadowMap = new QOpenGLFramebufferObject(w, h,
sFormat);
shadowMap->bind();
…
shadowMap->release();
m_program->setUniformValue("shadowMap", shadowMap-
>texture());
```

Frame Buffer Objects

- Also works with multiple render targets

That's all!