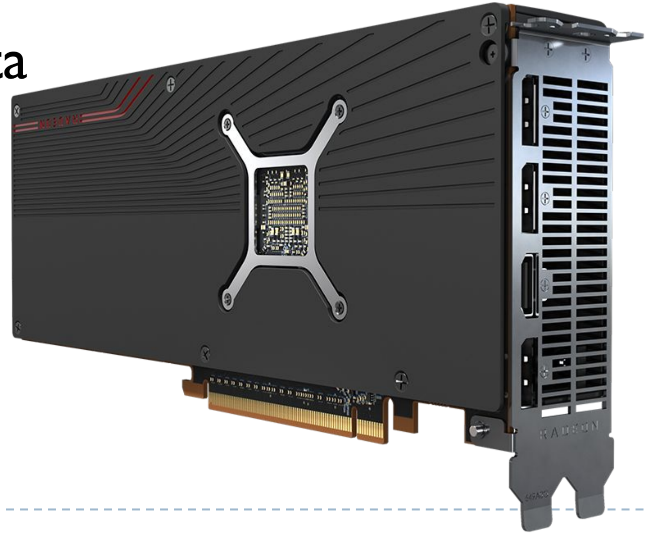
Introduction to Computer Graphics

- Background
- Rendering
- Graphics pipeline
- Rasterization
- **Graphics hardware and OpenGL**
  - **GPU & APIs**
  - **OpenGL Rendering pipeline**
  - **GLSL**
  - **Textures**
  - **Raster buffers**
- Human vision, colour & tone mapping

What is a GPU?

- Graphics Processing Unit
- Like CPU (Central Processing Unit) but for processing graphics
- Optimized for floating point operations on large arrays of data
  - Vertices, normals, pixels, etc.



What does a GPU do

- Performs all low-level tasks & a lot of high-level tasks
  - Clipping, rasterisation, hidden surface removal, …
    - Essentially draws millions of triangles very efficiently
  - Procedural shading, texturing, animation, simulation, …
  - Ray tracing (ray traversal, acceleration data structures)
  - Video rendering, de- and encoding, ...
  - Physics engines
- Full programmability at several pipeline stages
  - fully programmable
  - but optimized for massively parallel operations

What makes GPU so fast?

- 3D rendering can be very efficiently parallelized
  - Millions of pixels
  - Thousands of triangles
  - Many operations executed independently at the same time
- This is why modern GPUs
  - Contain between hundreds and thousands of SIMD processors
    - Single Instruction Multiple Data – operate on large arrays of data
  - >>1000 GB/s memory access
    - This is much higher bandwidth than CPU
    - But peak performance can be expected for very specific operations

GPU APIs (Application Programming Interfaces)

OpenGL

- Multi-platform
- Open standard API
- Focus on general 3D applications
  - Open GL driver manages the resources
- No ray tracing extensions

DirectX

- Microsoft Windows / Xbox
- Proprietary API
- Focus on games
  - Application manages resources

One more API

- Vulkan – cross platform, open standard
- Low-overhead API for high performance 3D graphics
- Compared to OpenGL / DirectX
  - Reduces CPU load
  - Better support of multi-CPU-core architectures
  - Finer control of GPU
- But
  - The code for drawing a few primitives can take 1000s line of code
  - Intended for game engines and code that must be very well optimized

And one more

- Metal (Apple iOS8)
  - low-level, low-overhead 3D GFX and compute shaders API
  - Support for Apple chips, Intel HD and Iris, AMD, Nvidia
  - Similar design as modern APIs, such as Vulcan
  - Swift or Objective-C API
  - Used mostly on iOS

GPGPU - general purpose computing

- OpenGL and DirectX are not meant to be used for general purpose computing
  - Example: physical simulation, machine learning
- CUDA – Nvidia's architecture for parallel computing
  - C-like programming language
  - With special API for parallel instructions
  - Requires Nvidia GPU
- OpenCL – Similar to CUDA, but open standard
  - Can run on both GPU and some CPUs
  - Supported by AMD, Intel and NVidia, Qualcomm, Apple, …

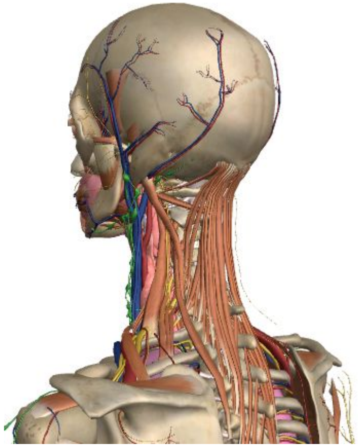
GPU and mobile devices

- OpenGL ES 1.0-3.2
  - Stripped version of OpenGL
  - Removed functionality that is not strictly necessary on mobile devices
- Devices
  - iOS: iPhone, iPad
  - Android phones
  - PlayStation 3
  - Nintendo 3DS
  - and many more

OpenGL ES 2.0 rendering (iOS)

WebGL and WebGPU

- WebGL (since ~2007)
  - JavaScript library for 3D rendering in a web browser
  - WebGL 1.0 - based on OpenGL ES 2.0
  - WebGL 2.0 – based on OpenGL ES 3.0
  - Used in 3D JavaScipt libraries
    - https://threejs.org/, WebXR
- WebGPU (since ~2017)
  - Provides access to Vulcan, Metal, DirectX 12
  - Own shading language WGSL (similar to Rust)



http://zygotebody.com/

OpenGL in Java

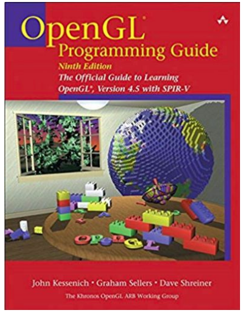
- Standard Java API does not include OpenGL interface
- But several wrapper libraries exist
  - Java OpenGL – JOGL
  - Lightweight Java Game Library - LWJGL
- We will use LWJGL 3
  - Seems to be better maintained
  - Access to other APIs (OpenCL, OpenAL, …)
- We also need a linear algebra library
  - JOML – Java OpenGL Math Library
  - Operations on 2, 3, 4-dimensional vectors and matrices

OpenGL History

- Proprietary library IRIS GL by SGI
- OpenGL 1.0 (1992)
- OpenGL 1.2 (1998)
- OpenGL 2.0 (2004)
  - GLSL
  - Non-power-of-two (NPOT) textures
- OpenGL 3.0 (2008)
  - Major overhaul of the API
  - Many features from previous versions depreciated
- OpenGL 3.2 (2009)
  - Core and Compatibility profiles
  - Geometry shaders
- OpenGL 4.0 (2010)
  - Catching up with Direct3D 11
- OpenGL 4.5 (2014)
- OpenGL 4.6 (2017)
  - SPIR-V shaders

How to learn OpenGL?

- Lectures – algorithms behind OpenGL, general principles
- Tick 2 – detailed tutorial, learning by doing
- References
  - OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.5 with SPIR-V by John Kessenich, Graham Sellers, Dave Shreiner ISBN-10: 0134495497
  - OpenGL quick reference guide https://www.opengl.org/documentation/glsl/
  - Google search: „man gl......"



OpenGL rendering pipeline

OpenGL programming model

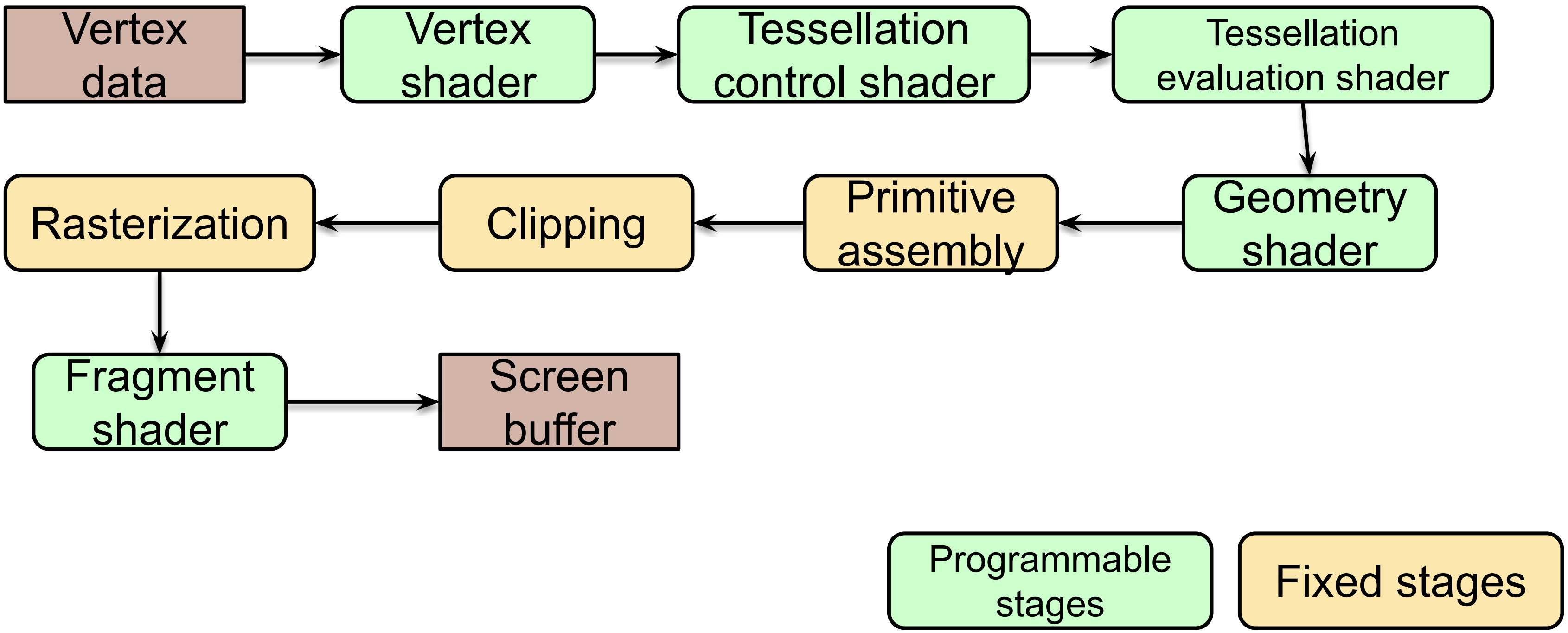
CPU code

- gl* functions that
  - Create OpenGL objects
  - Copy data CPU<->GPU
  - Modify OpenGL state
  - Enqueue operations
  - Synchronize CPU & GPU
- C99 library
- Wrappers in most programming language

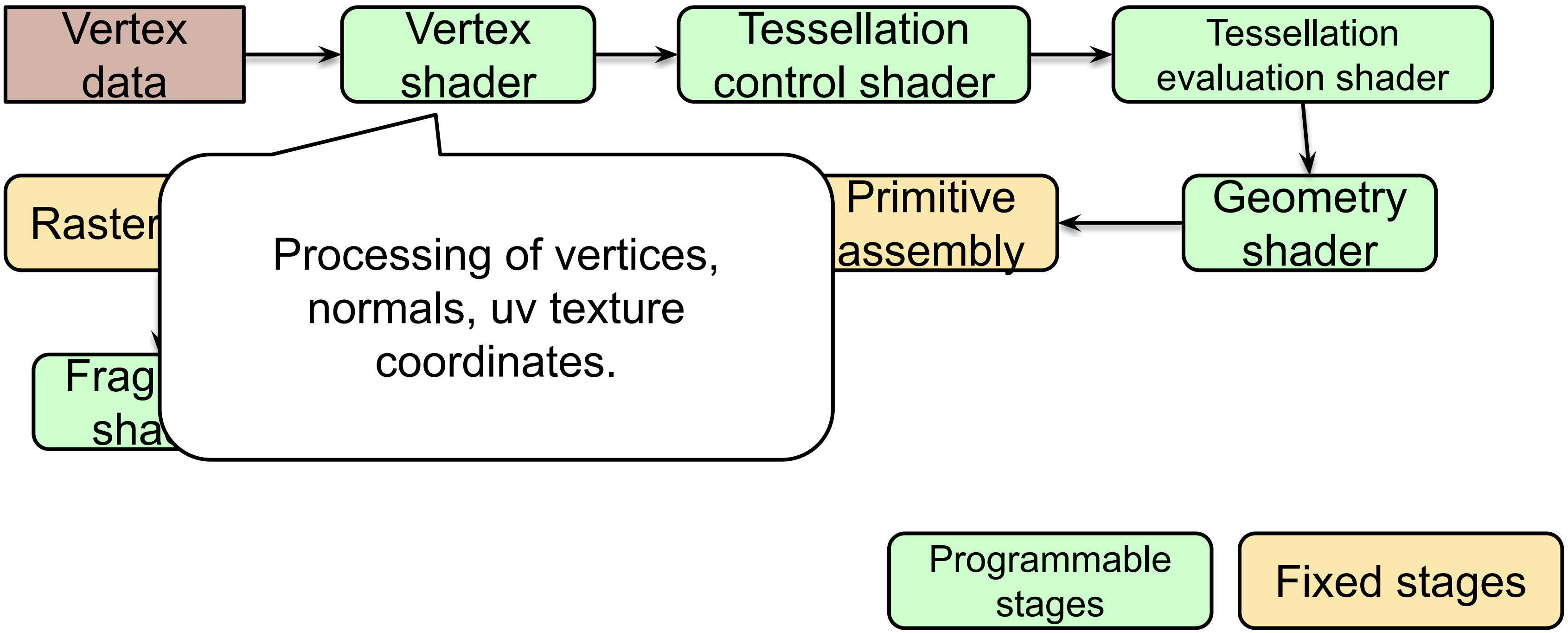
GPU code

- Fragment shaders
- Vertex shaders
- and other shaders
- Written in GLSL
  - Similar to C
  - From OpenGL 4.6 could be written in other language and compiled to SPIR-V

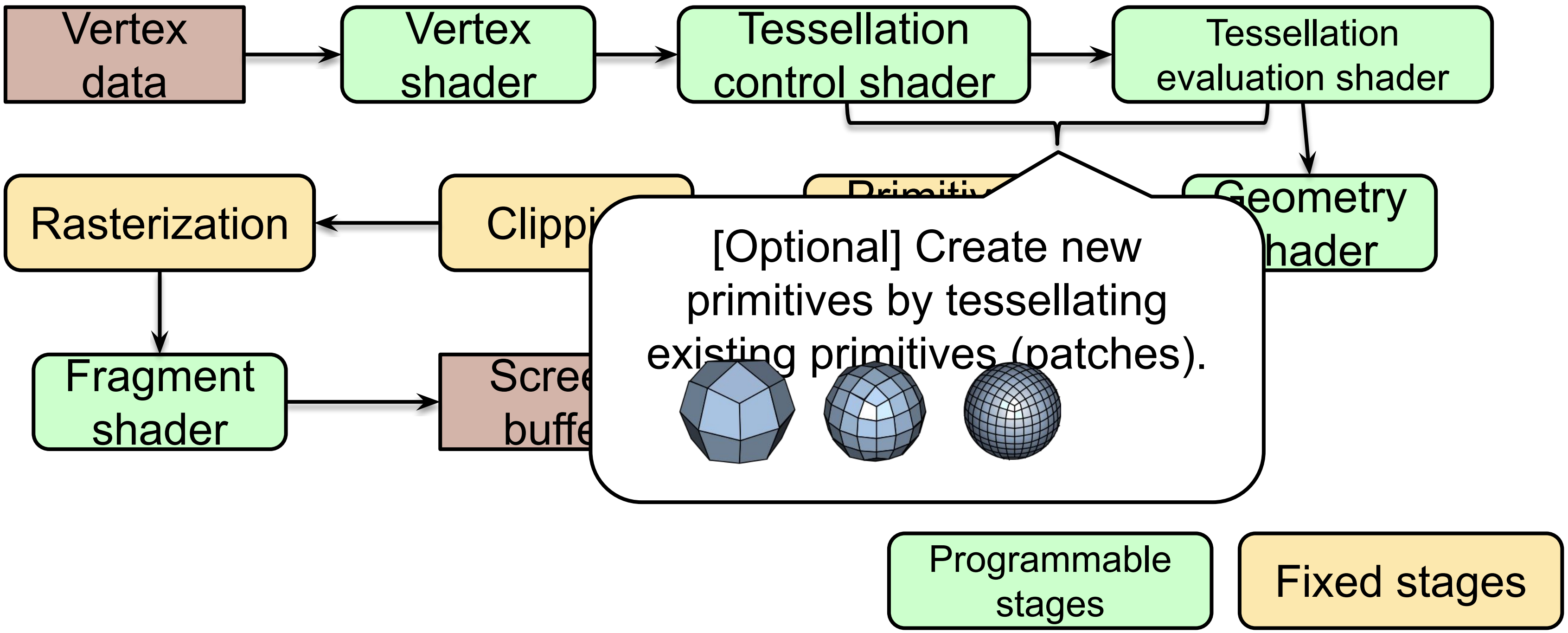
OpenGL rendering pipeline



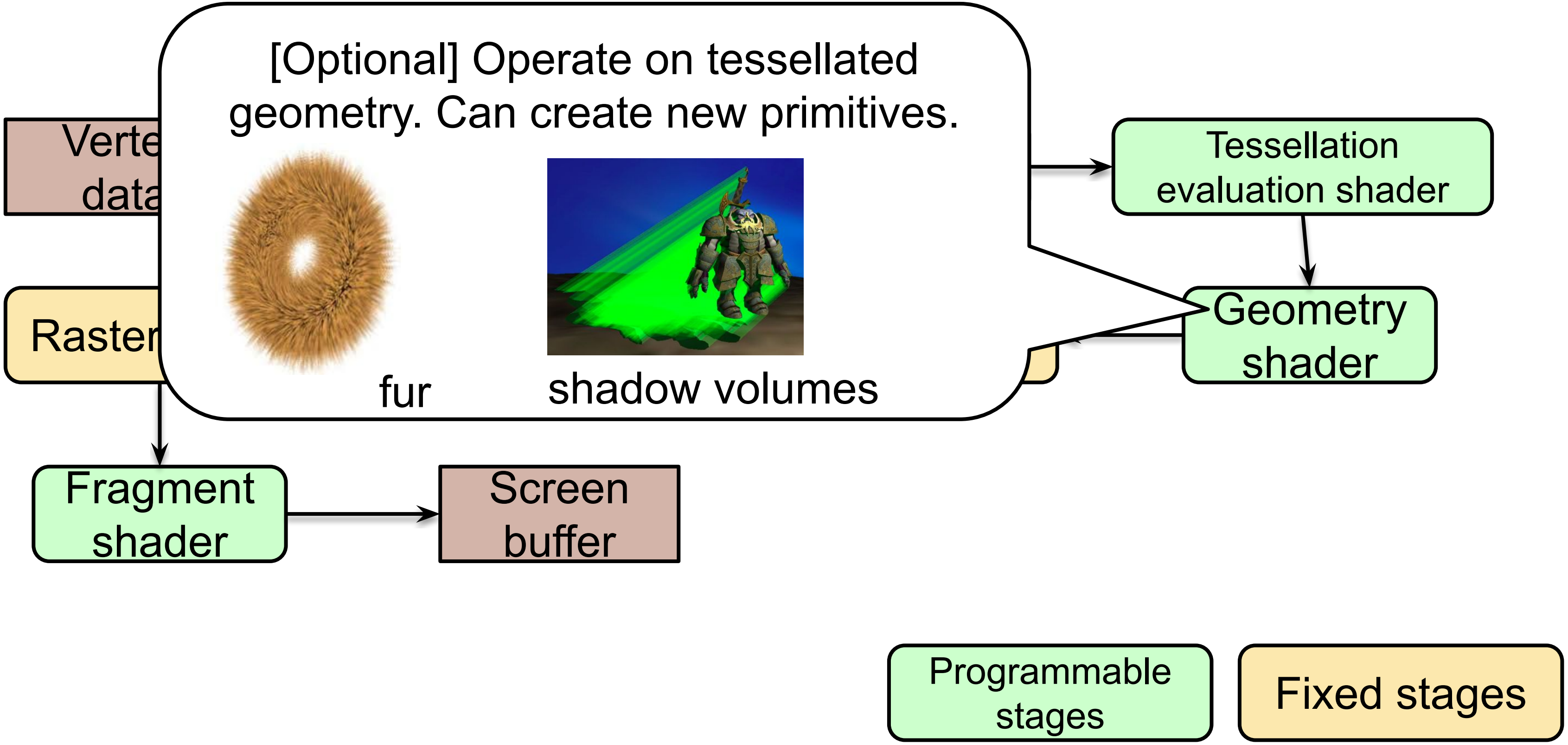
OpenGL rendering pipeline



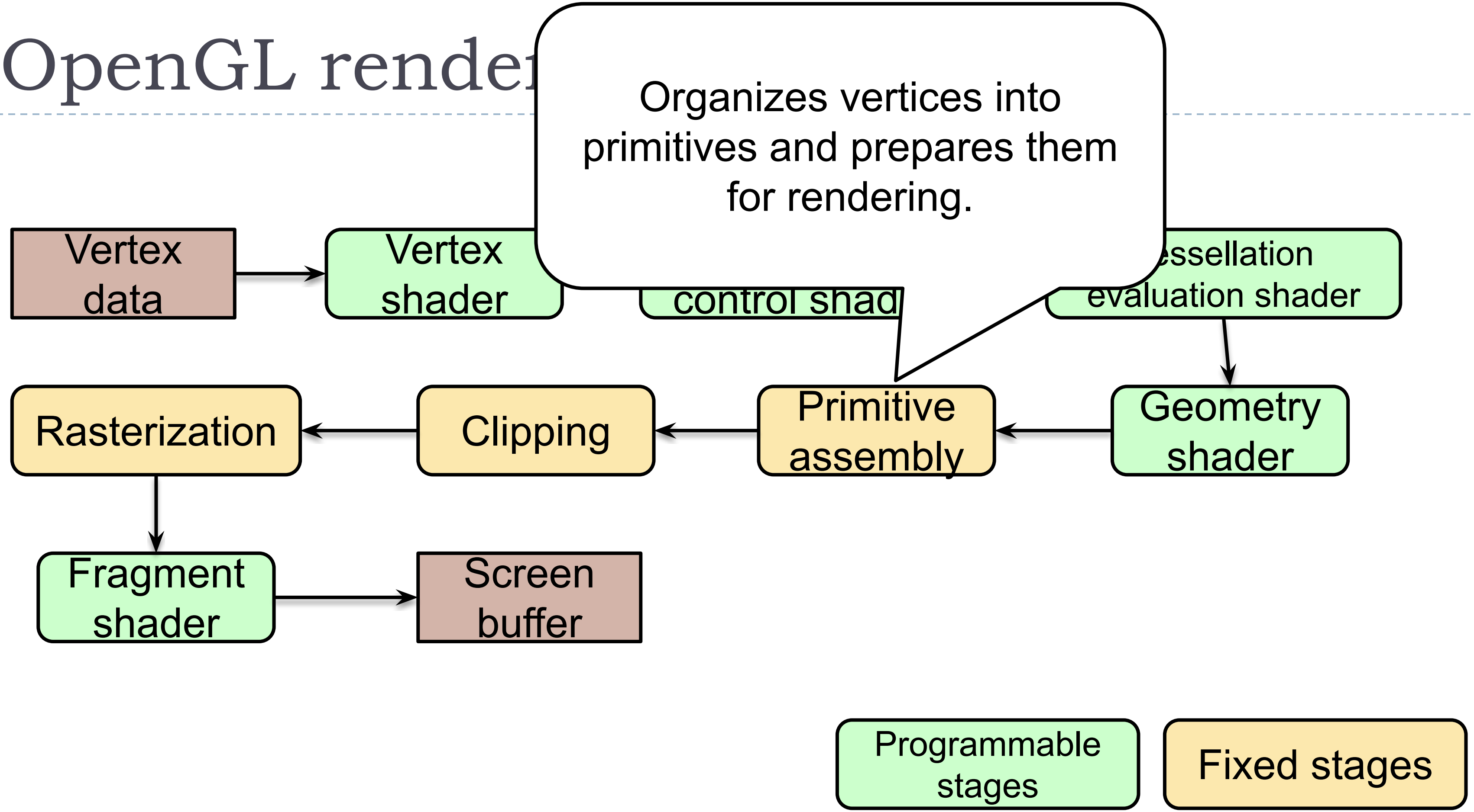
OpenGL rendering pipeline



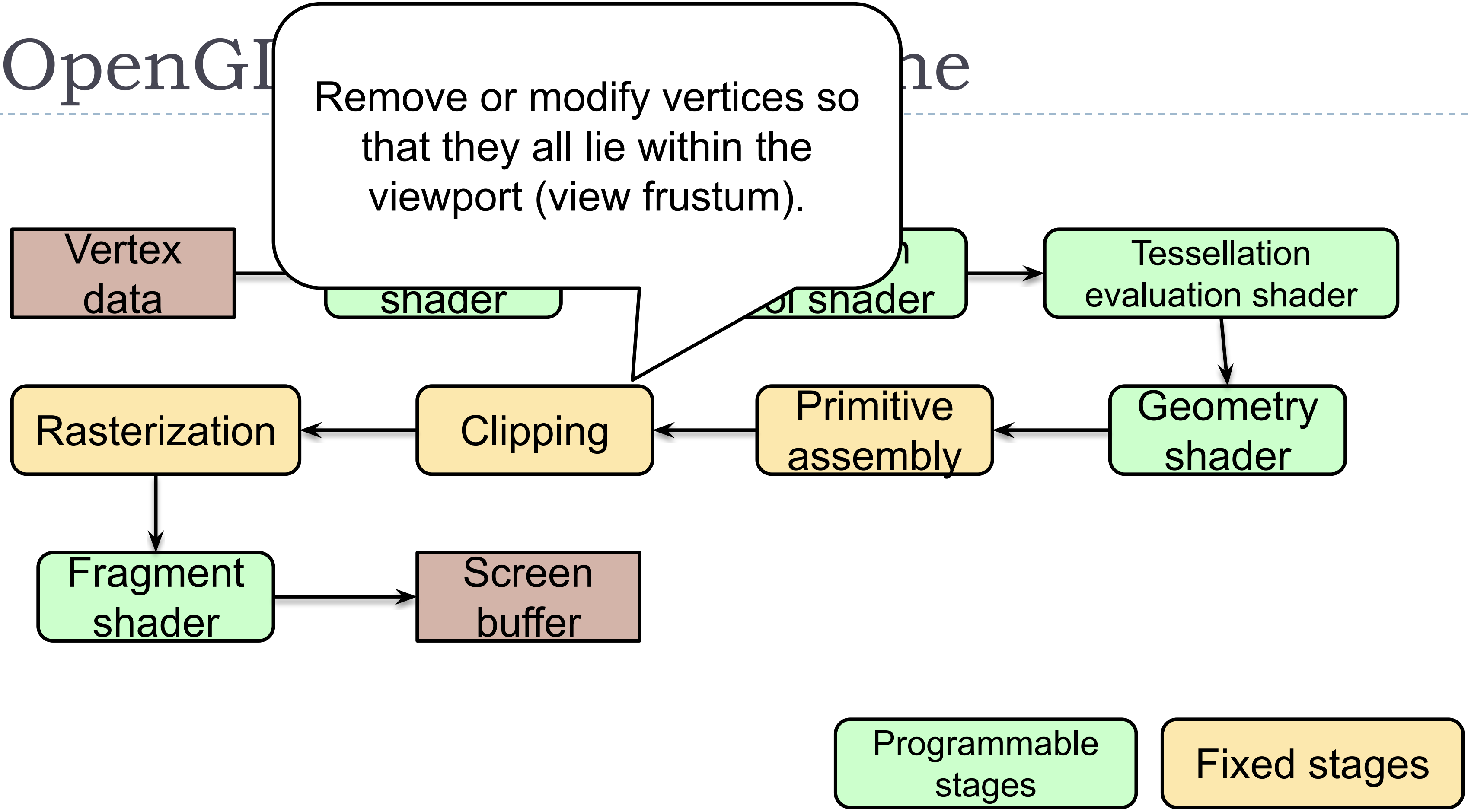
OpenGL rendering pipeline



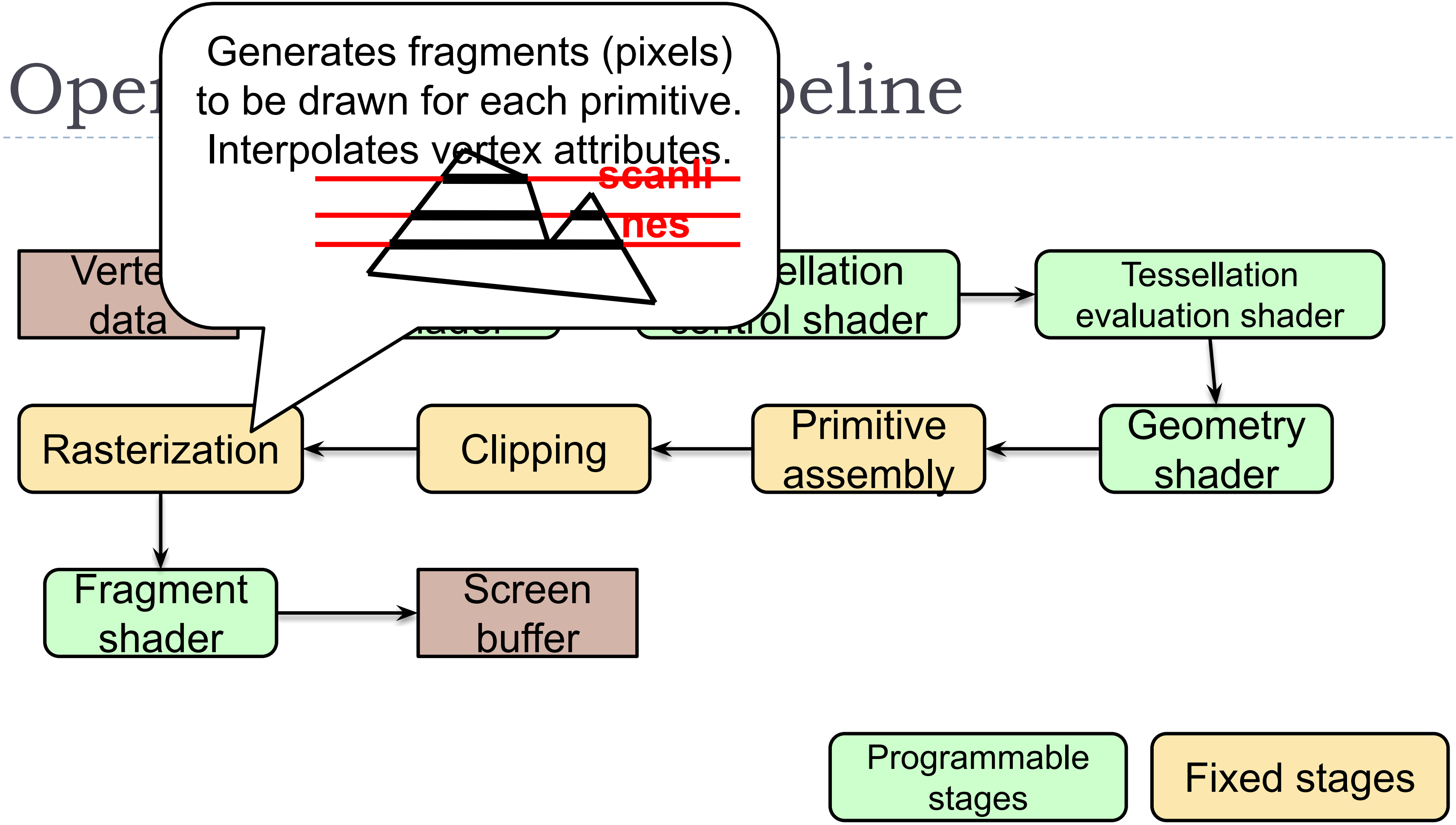
OpenGL rende



OpenGL ne

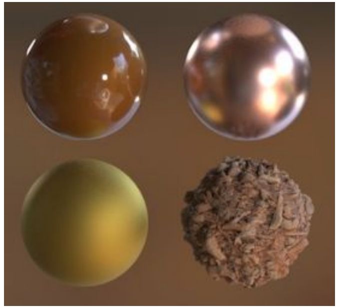


Remove or modify vertices so that they all lie within the viewport (view frustum).
Vertex data
shader
ol shader
Tessellation evaluation shader
Geometry shader
Primitive assembly
Clipping
Rasterization
Fragment shader
Screen buffer
Programmable stages
Fixed stages


Ope
peline
Generates fragments (pixels)
to be drawn for each primitive.
Interpolates vertex attributes.
scanli
nes
Verte
data
ellation
ol shader
Tessellation
evaluation shader
Geometry
shader
Primitive
assembly
Clipping
Rasterization
Fragment
shader
Screen
buffer
Programmable
stages
Fixed stages

OpenGL

Computes colour per each fragment (pixel). Can lookup colour in the texture. Can modify pixels' depth value.



Physically accurate materials

Non-Photorealistic-Rendering shader

Vertex data

Rasterization

Fragment shader

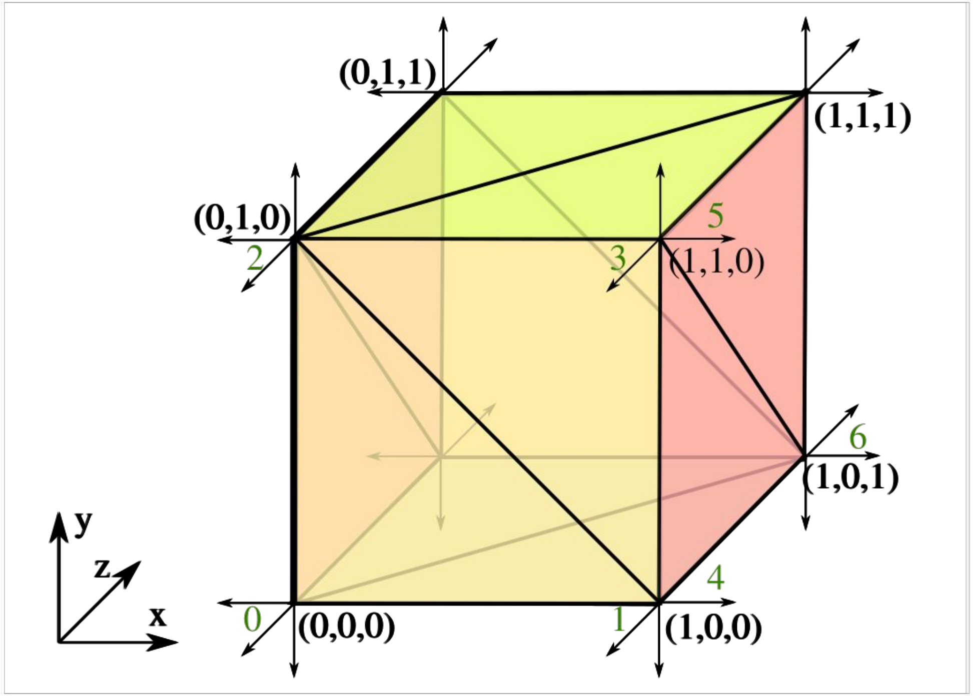
buffer

Also used for tone mapping.

Programmable stages

Fixed stages

Example: preparing vertex data for a cube



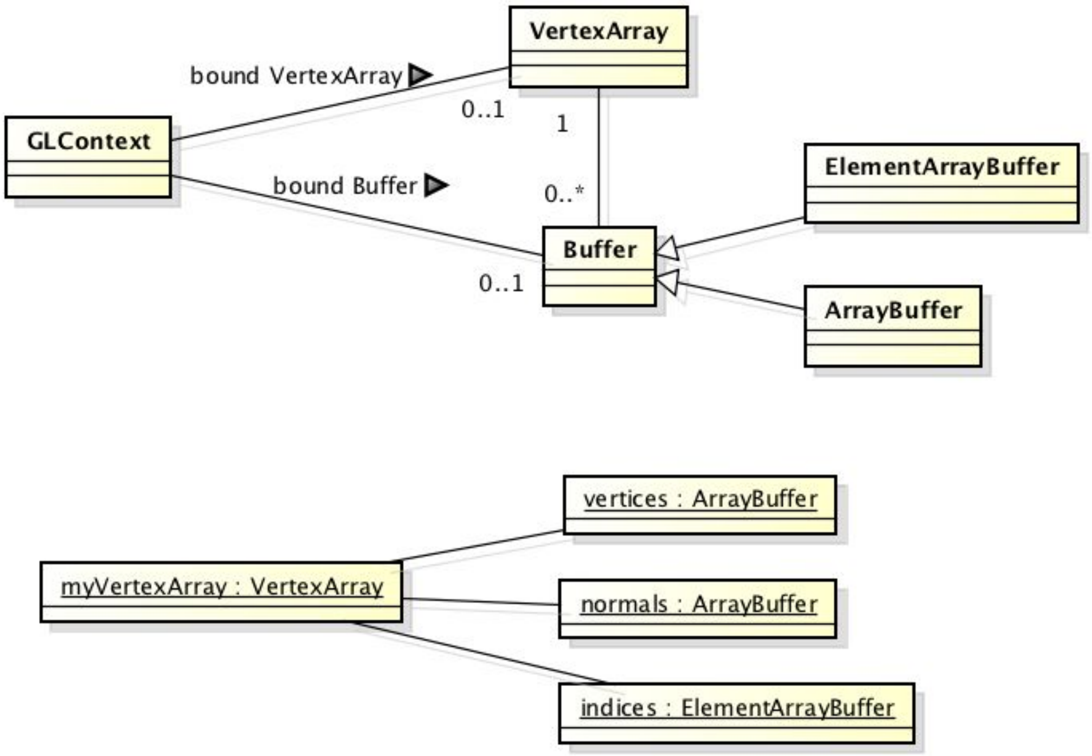
Primitives (triangles)

Indices
0, 1, 2
…

Vertex attributes

Ind	Positions	Normals
0	0, 0, 0	0, 0, -1
…	…	…

Geometry objects in OpenGL (OO view)



GLSL - fundamentals

Shaders

- Shaders are small programs executed on a GPU
  - Executed for each vertex, each pixel (fragment), etc.
- They are written in GLSL (OpenGL Shading Language)
  - Similar to C and Java
  - Primitive (int, float) and aggregate data types (ivec3, vec3)
  - Structures and arrays
  - Arithmetic operations on scalars, vectors and matrices
  - Flow control: if, switch, for, while
  - Functions

Example of a vertex shader

```
#version 330
in vec3 position;               // vertex position in local space
in vec3 normal;                 // vertex normal in local space
out vec3 frag_normal;                   // fragment normal in world space
uniform mat4 mvp_matrix;        // model-view-projection matrix

void main()
{
    // Typicaly normal is transformed by the model matrix
    // Since the model matrix is identity in our case, we do not modify normals
    frag_normal = normal;

    // The position is projected to the screen coordinates using mvp_matrix
    gl_Position = mvp_matrix * vec4(position, 1.0);
}
```

Why is this piece of code needed?

Data types

- Basic types
  - float, double, int, uint, bool
- Aggregate types
  - float: vec2, vec3, vec4; mat2, mat3, mat4
  - double: dvec2, dvec3, dvec4; dmat2, dmat3, dmat4
  - int: ivec2, ivec3, ivec4
  - uint: uvec2, uvec3, uvec4
  - bool: bvec2, bvec3, bvec4

```
vec3 V = vec3( 1.0, 2.0, 3.0 );
mat3 M = mat3( 1.0, 2.0, 3.0,
               4.0, 5.0, 6.0,
               7.0, 8.0, 9.0 );
```

Indexing components in aggregate types

- Subscripts: rgba, xyzw, stpq (work exactly the same)
  - `float red = color.r;`
  - `float v_y = velocity.y;`

  but also
  - `float red = color.x;`
  - `float v_y = velocity.g;`
- With 0-base index:
  - `float red = color[0];`
  - `float m22 = M[1][1];   // second row and column`

    `// of matrix M`

Swizzling

You can select the elements of the aggregate type:

```
vec4 rgba_color( 1.0, 1.0, 0.0, 1.0 );
vec3 rgb_color = rgba_color.rgb;
vec3 bgr_color = rgba_color.bgr;
vec3 grayscale = rgba_color.ggg;
```

Arrays

- Similar to C

```
float lut[5] = float[5]( 1.0, 1.42, 1.73, 2.0, 2.23 );
```

- Size can be checked with "`length()`"

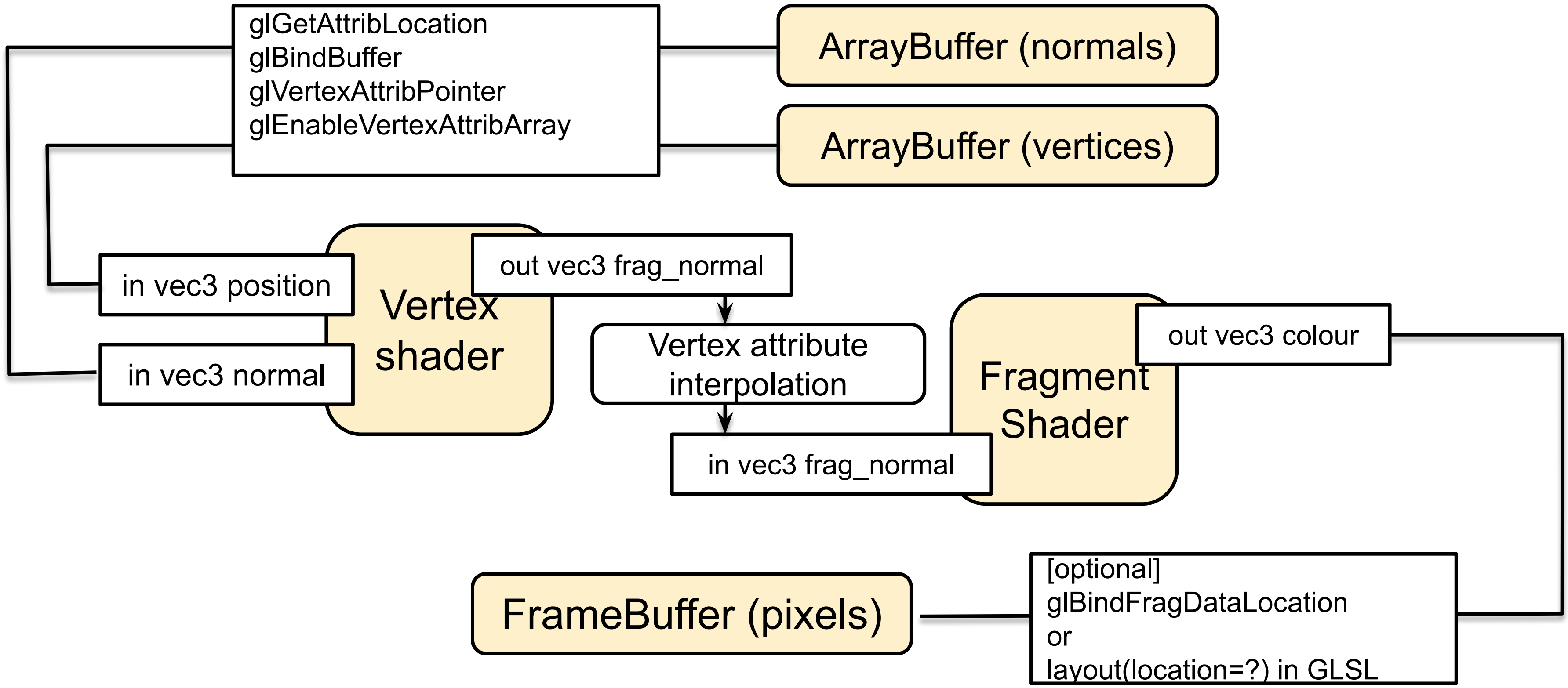
```
for( int i = 0; i < lut.length(); i++ ) {
   lut[i] *= 2;
}
```

Storage qualifiers

- `const` – read-only, fixed at compile time
- `in` – input to the shader
- `out` – output from the shader
- `uniform` – parameter passed from the application (Java), constant for the drawn geometry
- `buffer` – GPU memory buffer (allocated by the application), both read and write access
- `shared` – shared with a local work group (compute shaders only)

- Example: `const float pi=3.14;`

Shader inputs and outputs



GLSL Operators

- Arithmetic: + - ++ --
  - Multiplication:
    - `vec3 * vec3` – element-wise
    - `mat4 * vec4` – matrix multiplication (with a column vector)
- Bitwise (integer): `<<, >>, &, |, ^`
- Logical (bool): `&&, ||, ^^`
- Assignment:

```
float a=0;
a += 2.0; // Equivalent to a = a + 2.0
```

GLSL Math

- Trigonometric:
  - `radians( deg ), degrees( rad ), sin, cos, tan, asin, acos, atan, sinh, cosh, tanh, asinh, acosh, atanh`
- Exponential:
  - `pow, exp, log, exp2, log2, sqrt, inversesqrt`
- Common functions:
  - `abs, round, floor, ceil, min, max, clamp, …`
- Graphics
  - `reflect, refract, inversesqrt`
- And many more

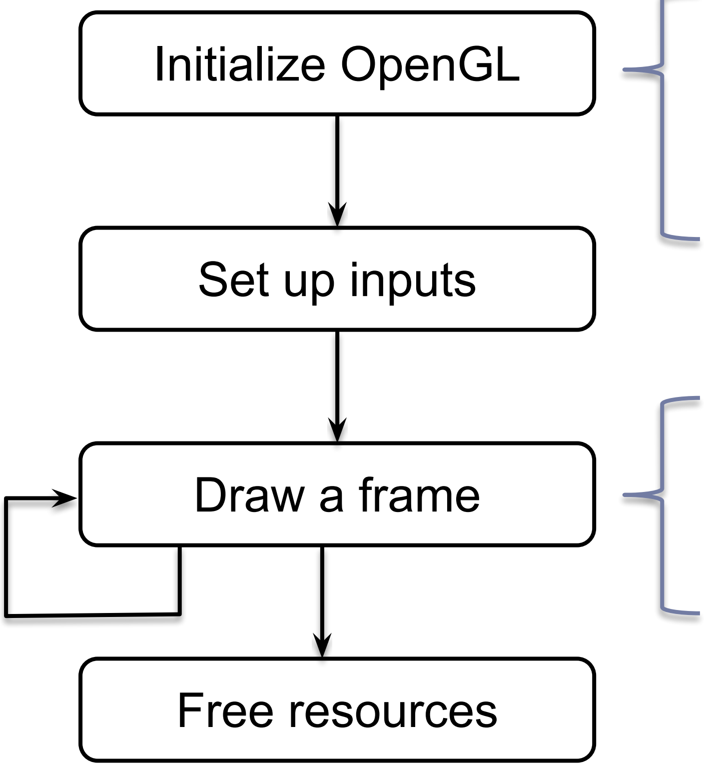
- See the quick reference guide at: https://www.opengl.org/documentation/glsl/

GLSL flow control

```
if( bool ) {
  // true
} else {
  // false
}

switch( int_value ) {
  case n:
    // statements
    break;
  case m:
    // statements
    break;
  default:
}
```

```
for( int i = 0; i<10; i++ ) {
   ...
}

while( n < 10 ) {
 ...
}

do {
 ...
} while ( n < 10 )
```

Simple OpenGL application - flow



- Initialize rendering window & OpenGL context
- Send the geometry (vertices, triangles, normals) to the GPU
- Load and compile Shaders

- Clear the screen buffer
- Set the model-view-projection matrix
- Render geometry
- Flip the screen buffers

Rendering geometry

- To render a single object with OpenGL

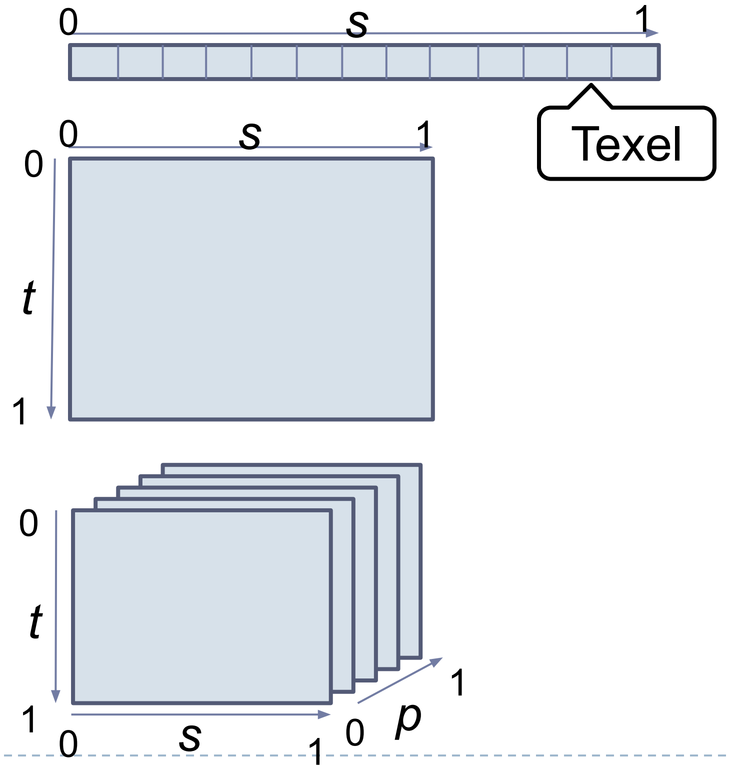
1. `glUseProgram()` – to activate vertex & fragment shaders
2. `glVertexAttribPointer()` – to indicate which Buffers with vertices and normals should be input to the vertex shader
3. `glUniform*()` – to set uniforms (parameters of the fragment/vertex shader)
4. `glBindTexture()` – to bind the texture
5. `glBindVertexArray()` – to bind the vertex array
6. `glDrawElements()` – to queue drawing the geometry
7. Unbind all objects

- OpenGL API is designed around the idea of a state-machine – set the state & queue drawing command

Textures

(Most important) OpenGL texture types

1D

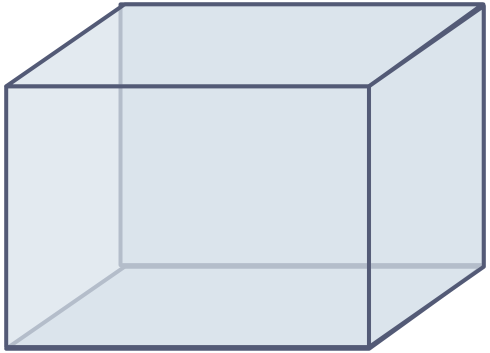


2D

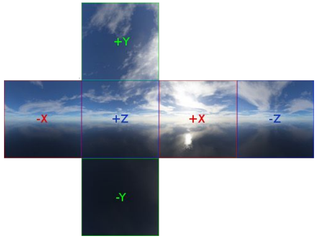
3D

Texture can have any size but the sizes that are powers of two (POT, 2^n) may give better performance.

CUBE_MAP

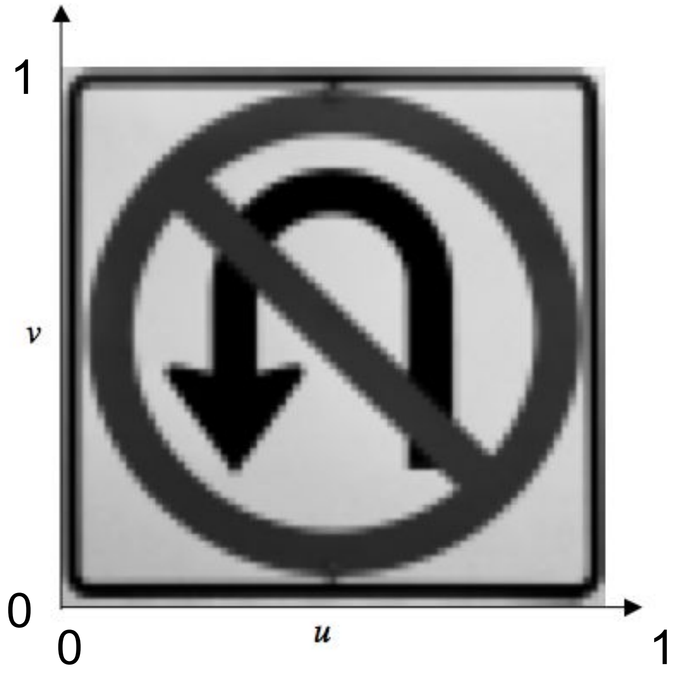


Used for environment mapping



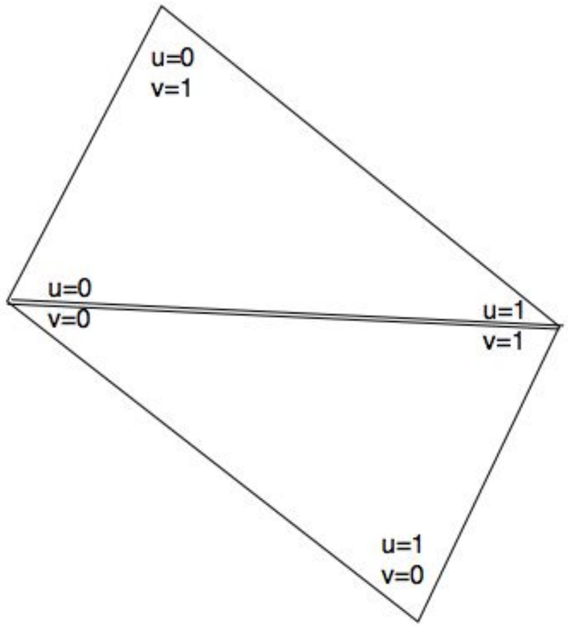
Texture mapping

- 1. Define your texture function (image) T(u,v)
- (u,v) are texture coordinates



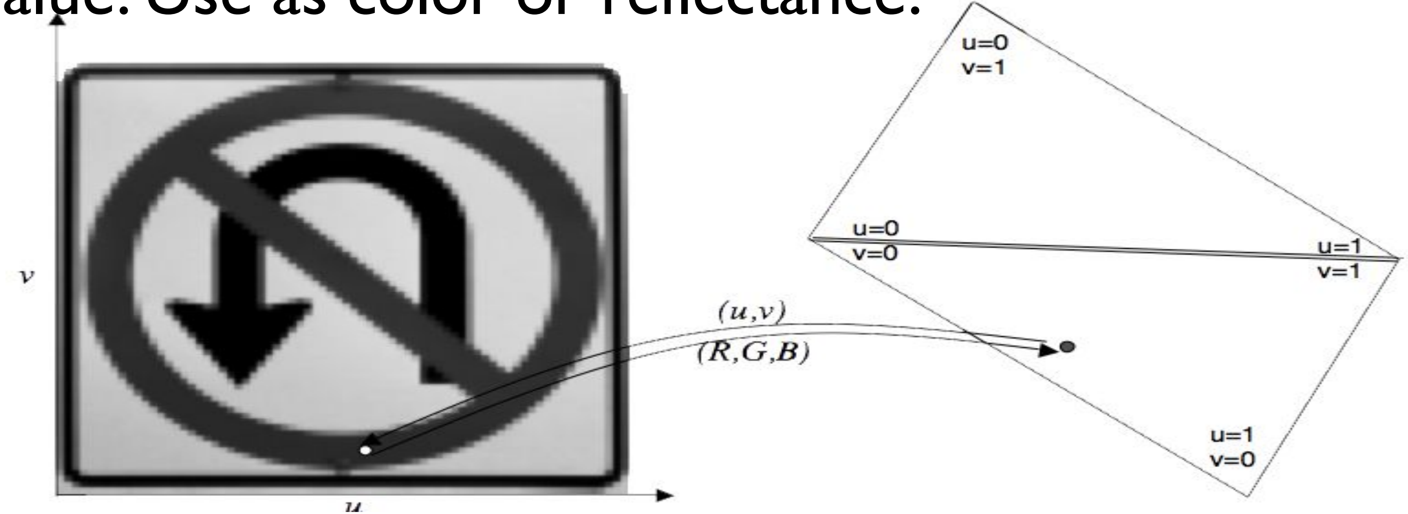
Texture mapping

- 2. Define the correspondence between the vertices on the 3D object and the texture coordinates

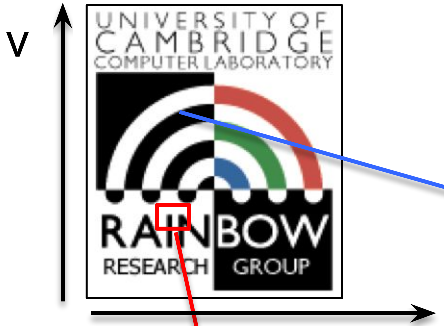


Texture mapping

- 3. When rendering, for every surface point compute texture coordinates. Use the texture function to get texture value. Use as color or reflectance.

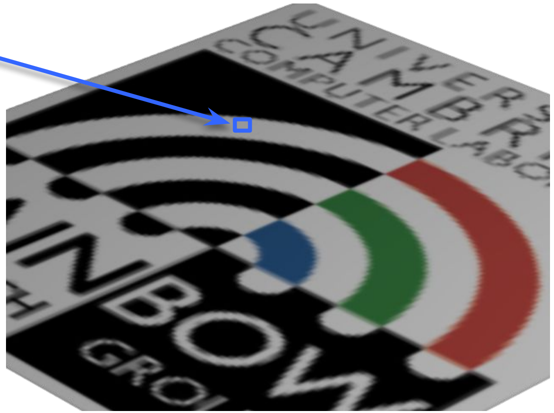


Sampling



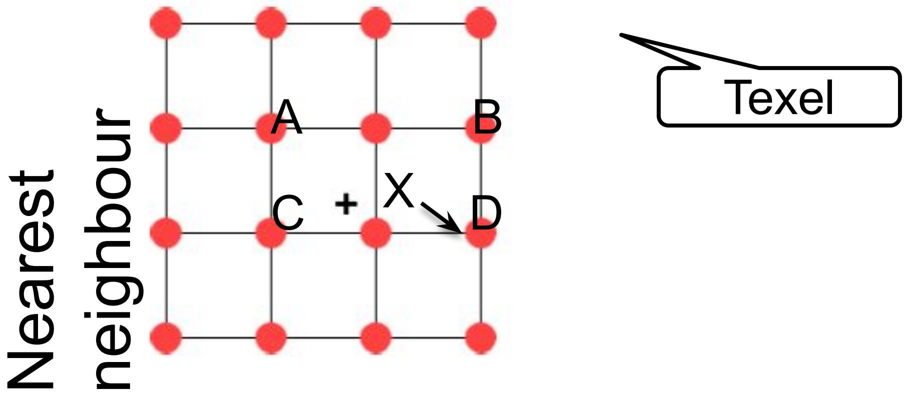
Texture

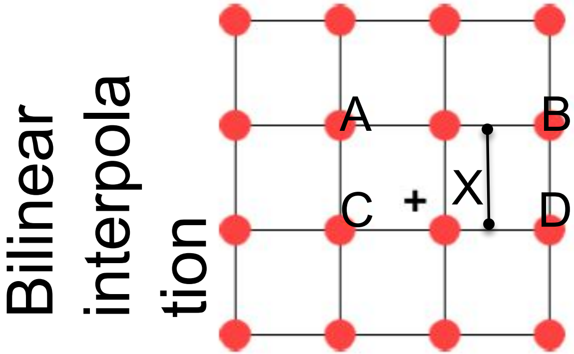
Up-sampling
More pixels than texels
Values need to be interpolated



Down-sampling
Fewer pixels than texels
Values need to be averaged over an area of the texture (usually using a mipmap)

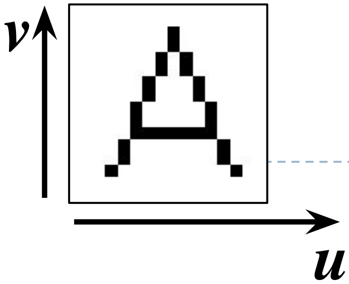
Nearest neighbor vs. bilinear interpolation (upsampling)



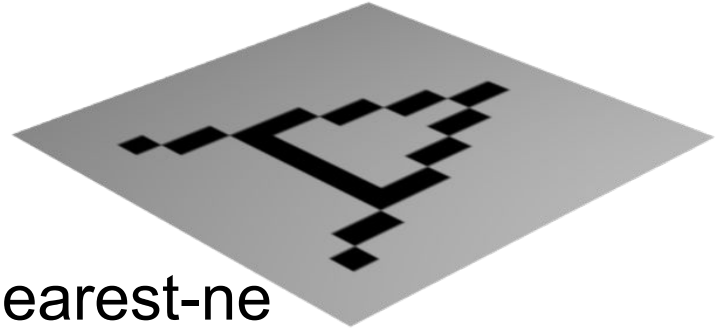


Pick the nearest texel: D

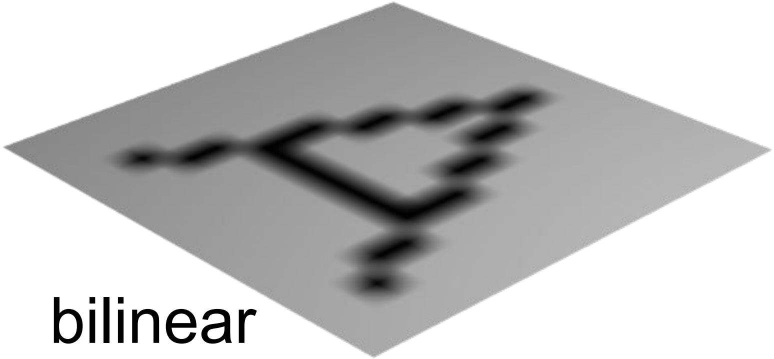
Interpolate first along x-axis between AB and CD, then along y-axis between the interpolated points.



Texture mapping examples



nearest-neighbour

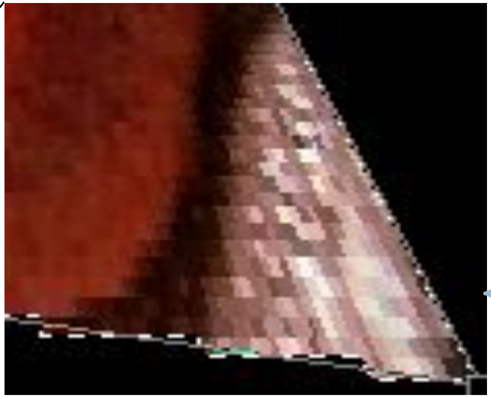


bilinear

Up-sampling

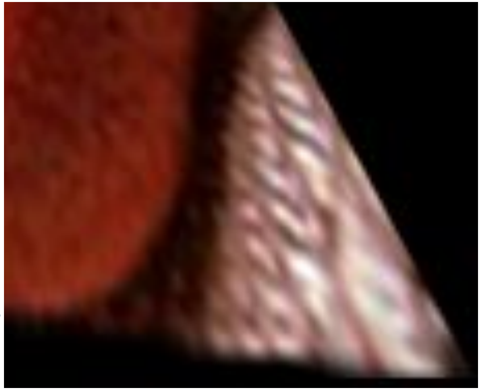
nearest-neighbour

blocky artefacts



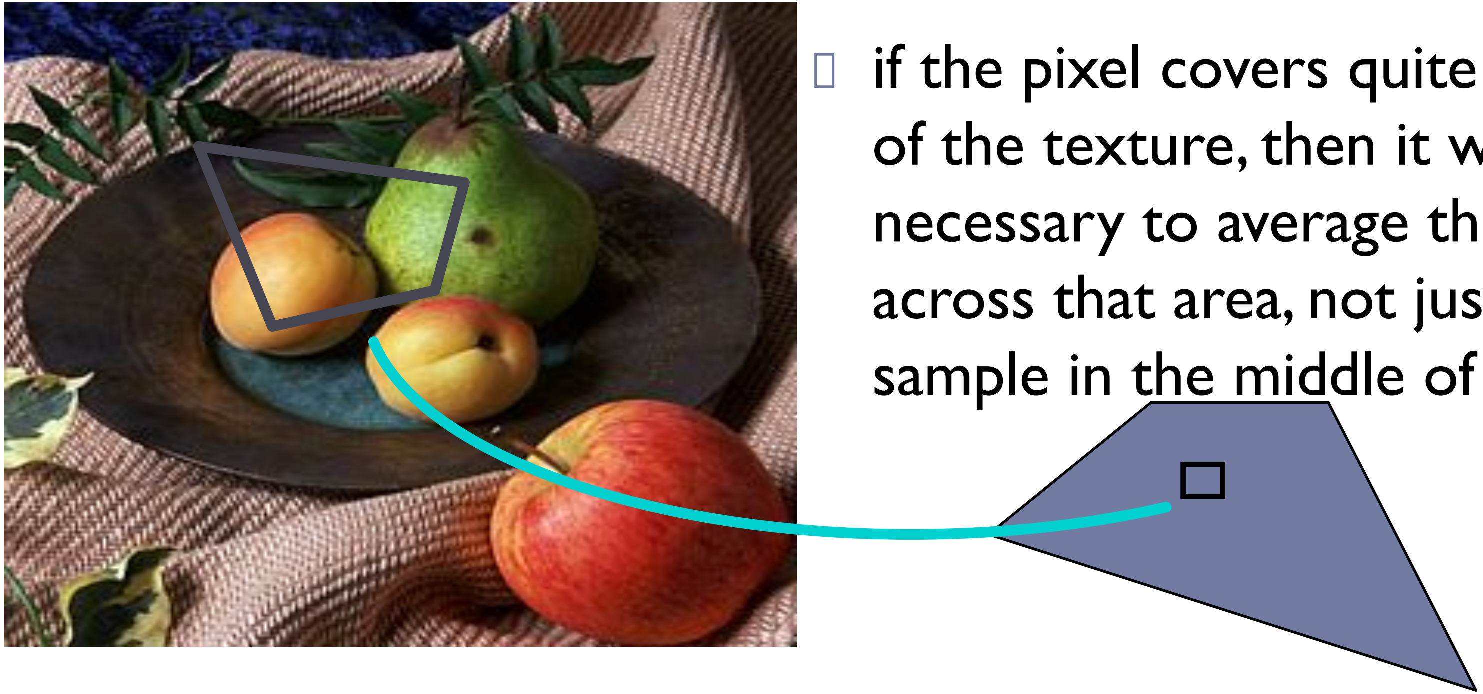
bilinear

blurry artefacts



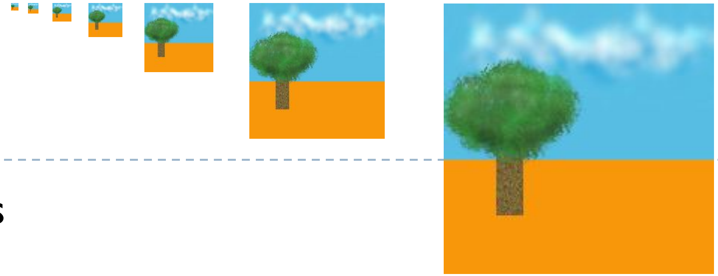
- if one pixel in the texture map covers several pixels in the final image, you get visible artefacts
- only practical way to prevent this is to ensure that texture map is of sufficiently high resolution that it does not happen

Down-sampling

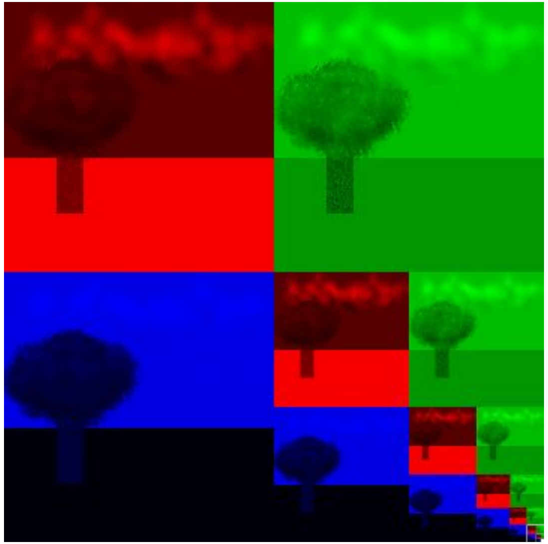


- if the pixel covers quite a large area of the texture, then it will be necessary to average the texture across that area, not just take a sample in the middle of the area

Mipmap



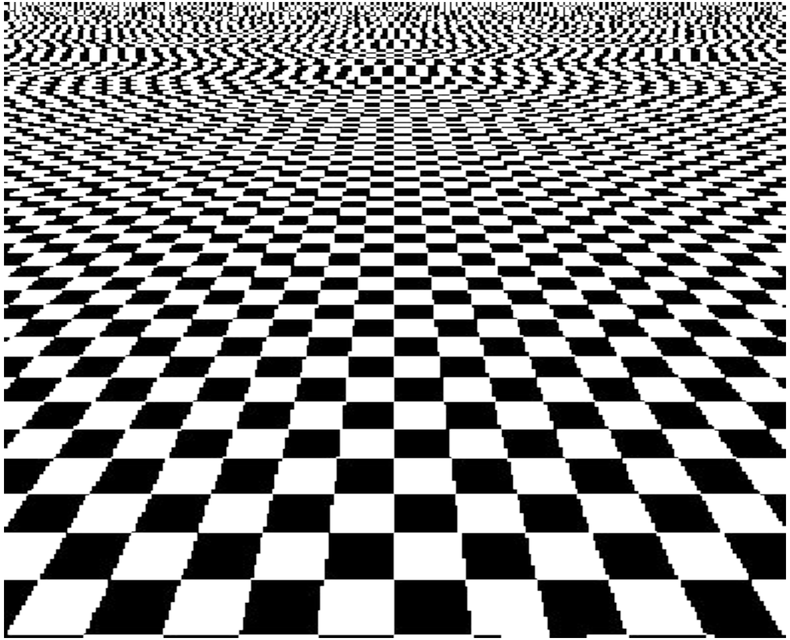
- Textures are often stored at multiple resolutions as a mipmap
  - Each level of the pyramid is half the size of the lower level
  - Mipmap resolution is always power-of-two (1024, 512, 256, 128, ...)
- It provides pre-filtered texture (area-averaged) when screen pixels are larger than the full resolution texels
- Mipmap requires just an additional 1/3 of the original texture size to store
- OpenGL can generate a mipmap with `glGenerateMipmap(GL_TEXTURE_2D)`



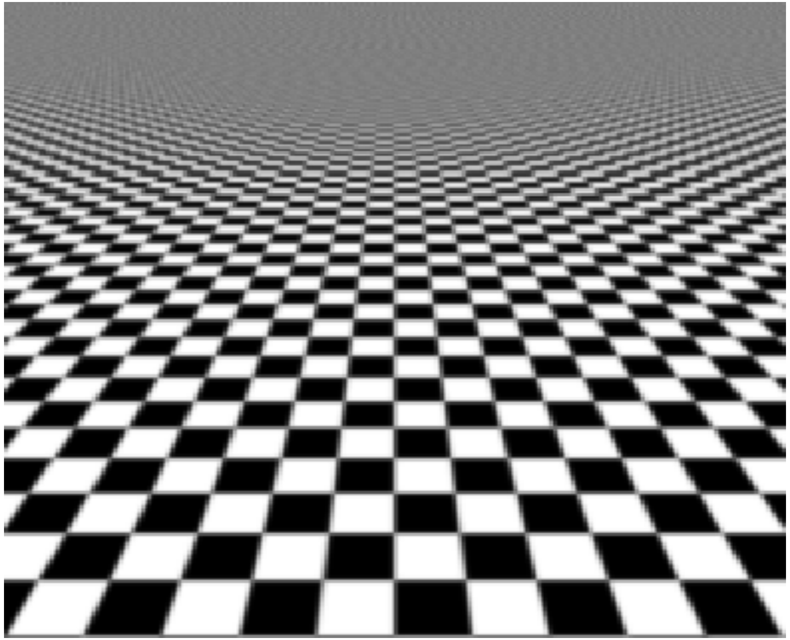
This image is an illustration showing only 1/3 increase in storeage. Mipmaps are stored differently in the GPU memory.

Down-sampling

without area averaging

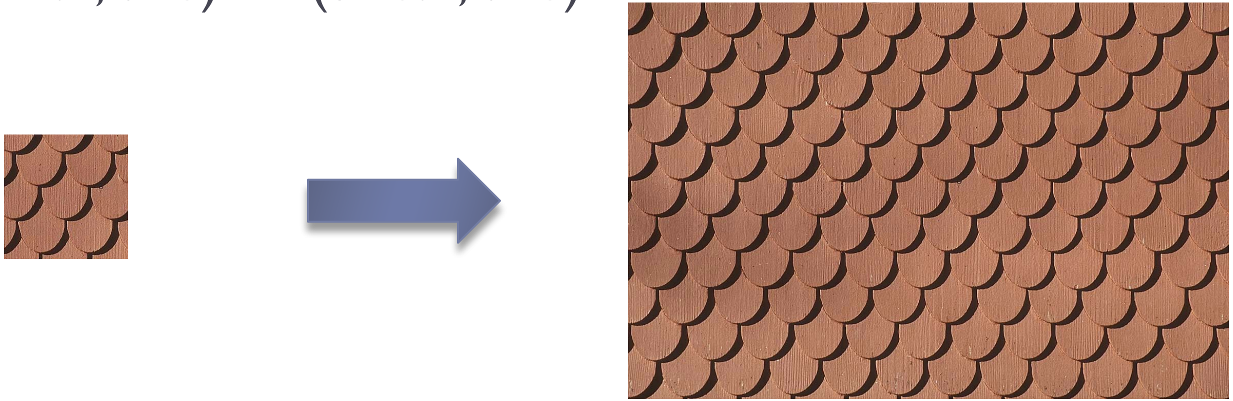


with area averaging



Texture tiling

- Repetitive patterns can be represented as texture tiles.
- The texture folds over, so that
  - $T(u=1.1, v=0) = T(u=0.1, v=0)$



Gimp and other drawing software often offer plugins for creating tiled textures

Multi-surface UV maps

- A single texture is often used for multiple surfaces and objects

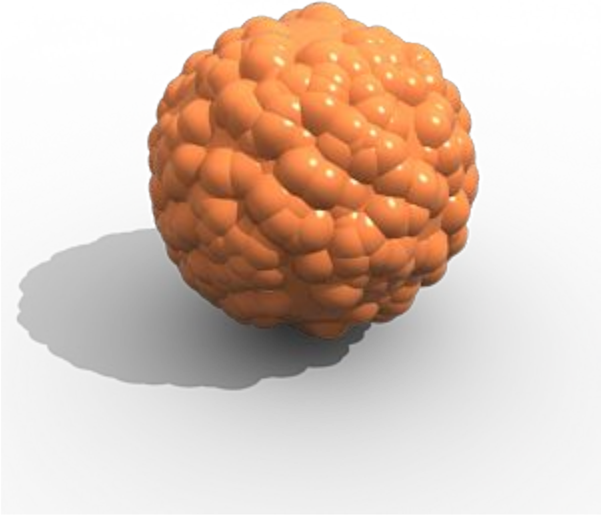
Example from:
http://awshub.com/blog/blog/2011/11/01/hi-poly-vs-low-poly/

Bump mapping and normal mapping

- Special kind of texture that modifies surface normal
  - Surface normal is a vector that is perpendicular to a surface
- The surface is still flat but shading appears as on an uneven surface
- Easily done in fragment shaders

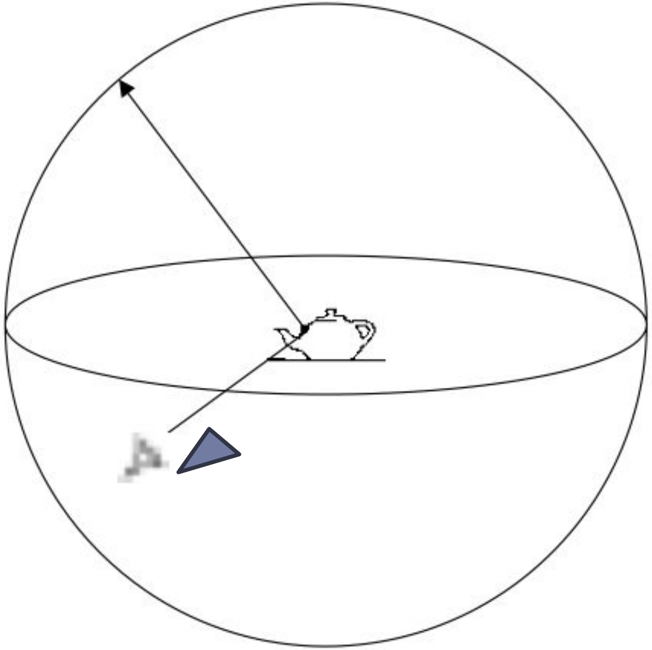
Displacement mapping

- Texture that modifies surface
- Better results than bump mapping since the surface is not flat
- Requires geometry shaders



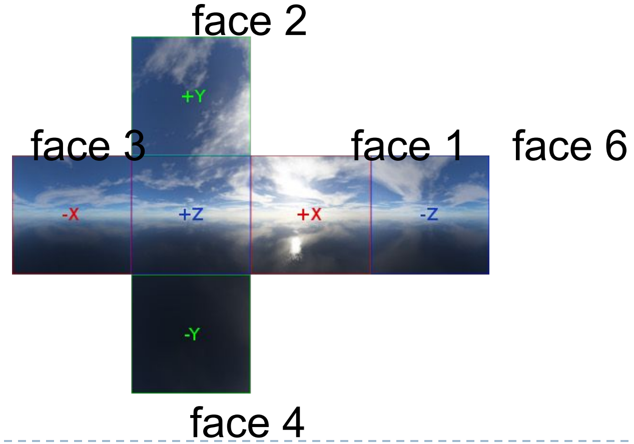
Environment mapping

- To show environment reflected by an object
  - Assumption: infinite distance to the source of reflection

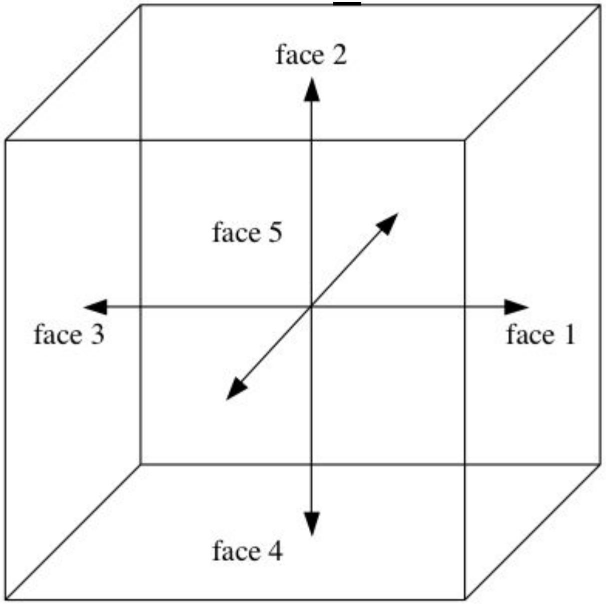


Environment mapping

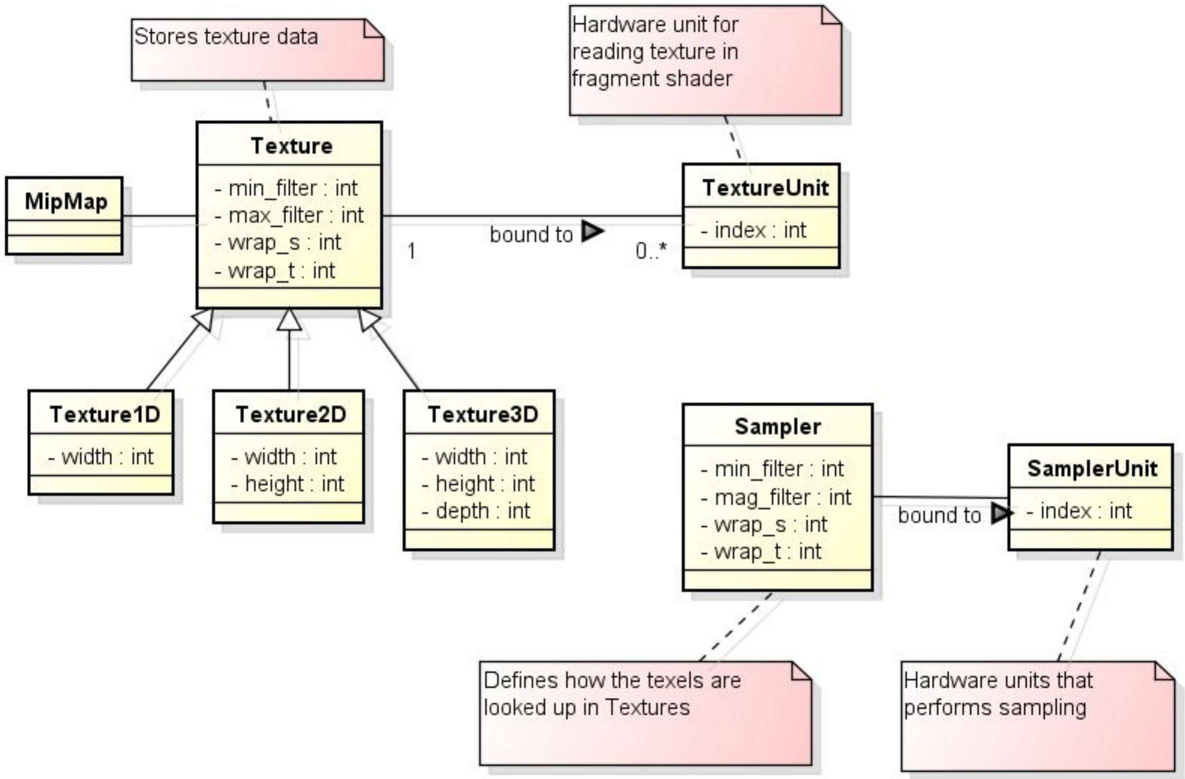
- Environment cube
- Each face captures environment in that direction



CUBE_MAP



Texture objects in OpenGL



Texture parameters

```
//Setup filtering, i.e. how OpenGL will interpolate the pixels when scaling up or down
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_NEAREST);
```

How to interpolate in 2D

How to interpolate between mipmap levels

```
//Setup wrap mode, i.e. how OpenGL will handle pixels outside of the expected range
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
```

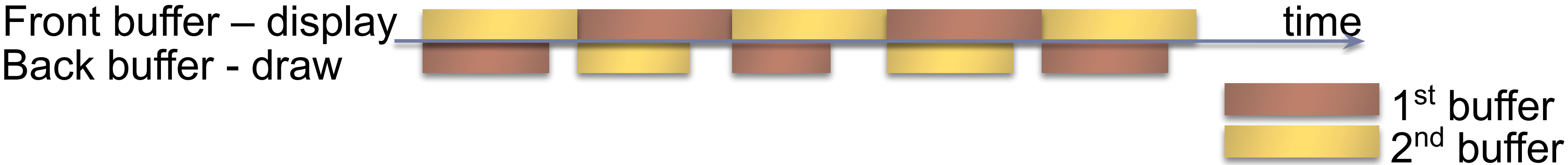
Raster buffers (colour, depth, stencil)

Render buffers in OpenGL

Colour:	GL_FRONT	GL_BACK	Four components: RGBA
In stereo:	GL_FRONT_LEFT	GL_FRONT_RIGHT	Typically 8 bits per component
	GL_BACK_LEFT	GL_BACK_RIGHT	
Depth:	DEPTH		To resolve occlusions (see Z-buffer algorithm) Single component, usually >8 bits
Stencil:	STENCIL		To block rendering selected pixels Single component, usually 8 bits.

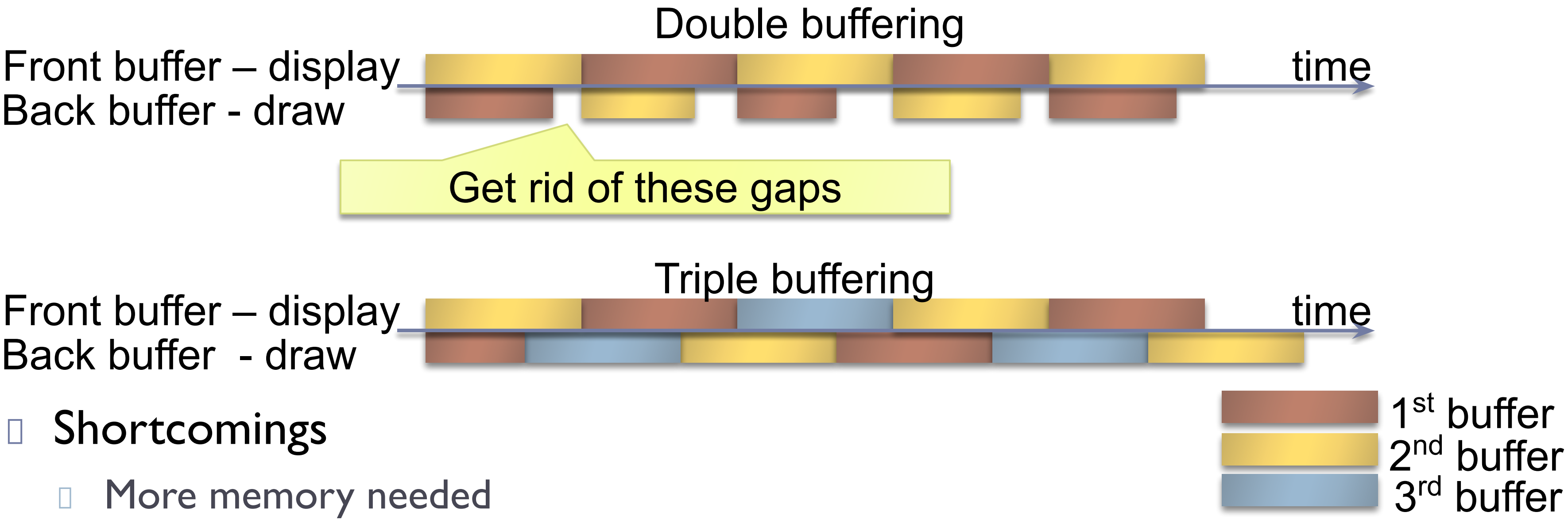
Double buffering

- To avoid flicker, tearing
- Use two buffers (rasters):
  - Front buffer – what is shown on the screen
  - Back buffer – not shown, GPU draws into that buffer
- When drawing is finished, swap front- and back-buffers



Triple buffering

- Do not wait for swapping to start drawing the next frame



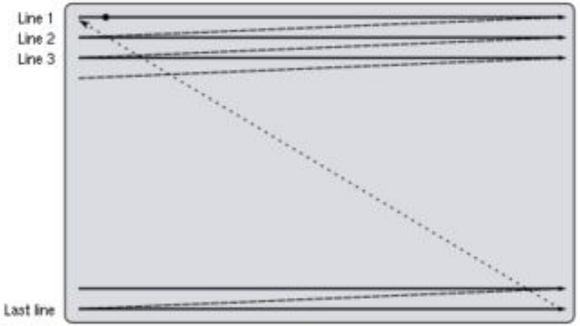
- Shortcomings
  - More memory needed
  - Higher delay between drawing and displaying a frame

Vertical Synchronization: V-Sync

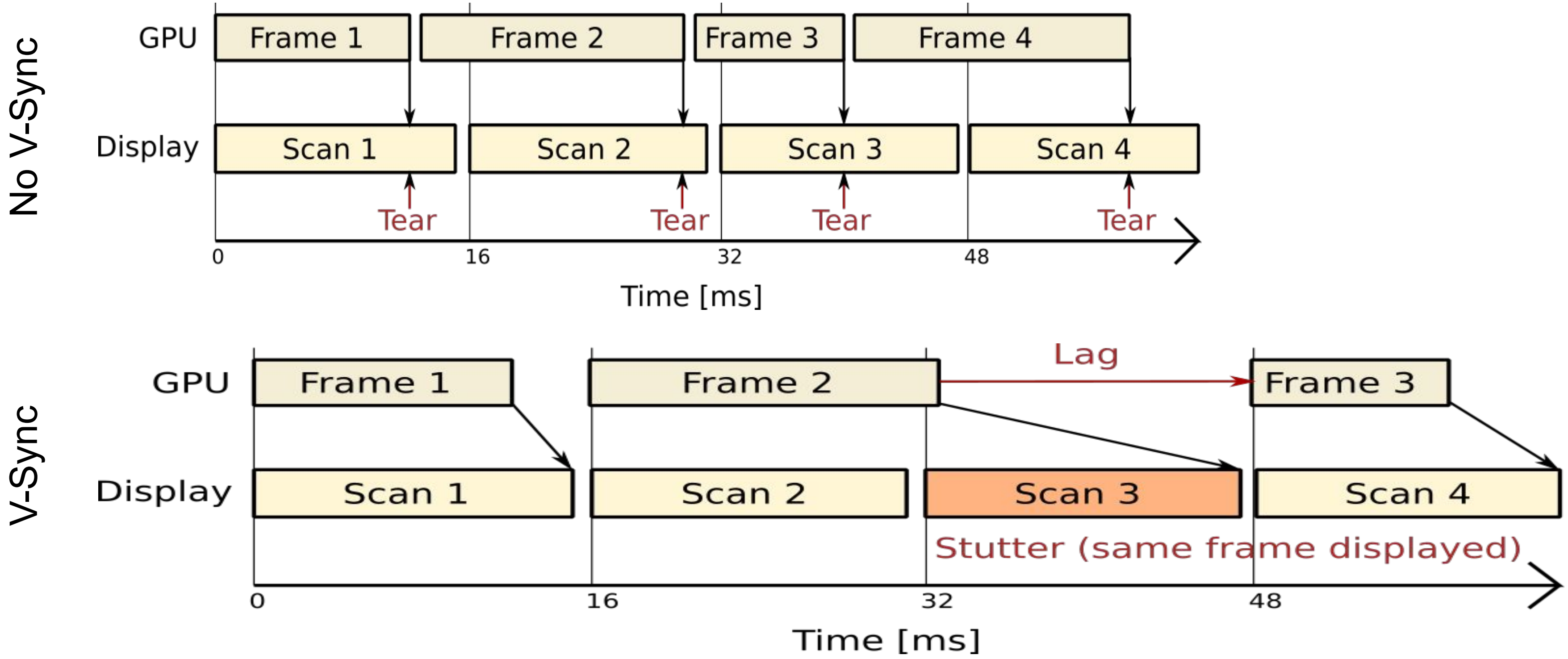
- Pixels are copied from colour buffer to monitor row-by-row
- If front & back buffer are swapped during this process:
  - Upper part of the screen contains previous frame
  - Lower part of the screen contains current frame
  - Result: tearing artefact

- Solution: When V-Sync is enabled
  - `glwfSwapInterval(1);`

`glSwapBuffers()` waits until the last row of pixels is copied to the display.



No V-Sync vs. V-Sync



FreeSync (AMD) & G-Sync (Nvidia)

- Adaptive sync or Variable Refresh Rate (VRR)
  - Graphics card controls timing of the frames on the display
  - Can save power for 30fps video of when the screen is static
  - Can reduce lag for real-time graphics