Visualization

Lectures on Modern Scientific Programming 2016

Dániel Berényi
Wigner GPU Lab

Overview

In this lecture we will cover the following main cases and differences:

- Vector vs Raster
- Software vs Hardware
- Off-line vs real-time
- 2D vs 3D

- Mathematics

Image representation

Images can be represented in two remarkably different ways:

- Vector
- Raster

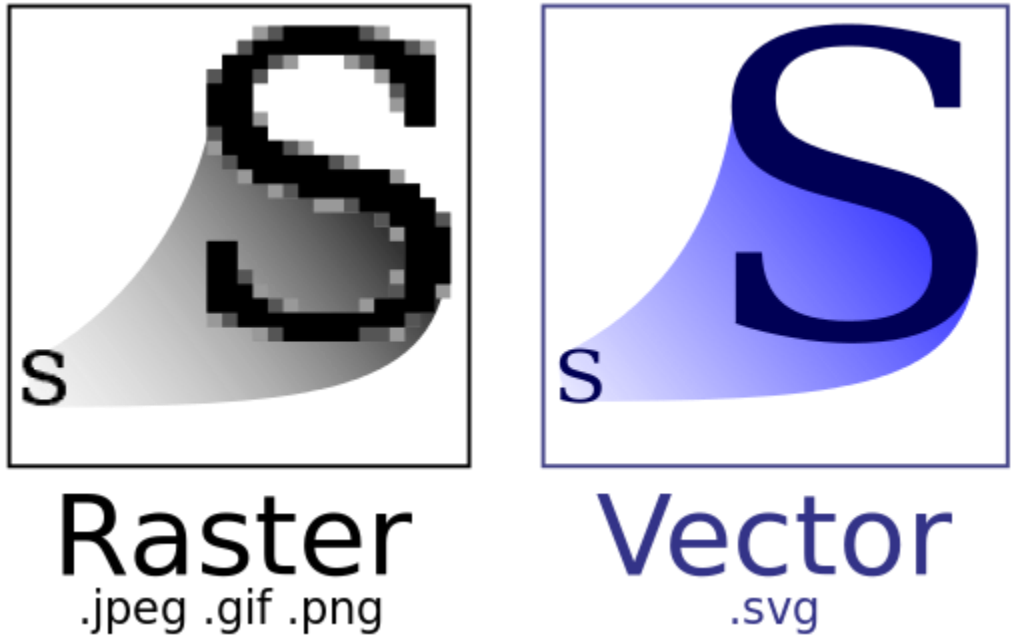


Image representation

Images can be represented in two remarkably different ways:

- Vector
  - Represents the objects and effects on their own right that make up the image
- Raster
  - Represents the final result of a possibly very complex drawing operation

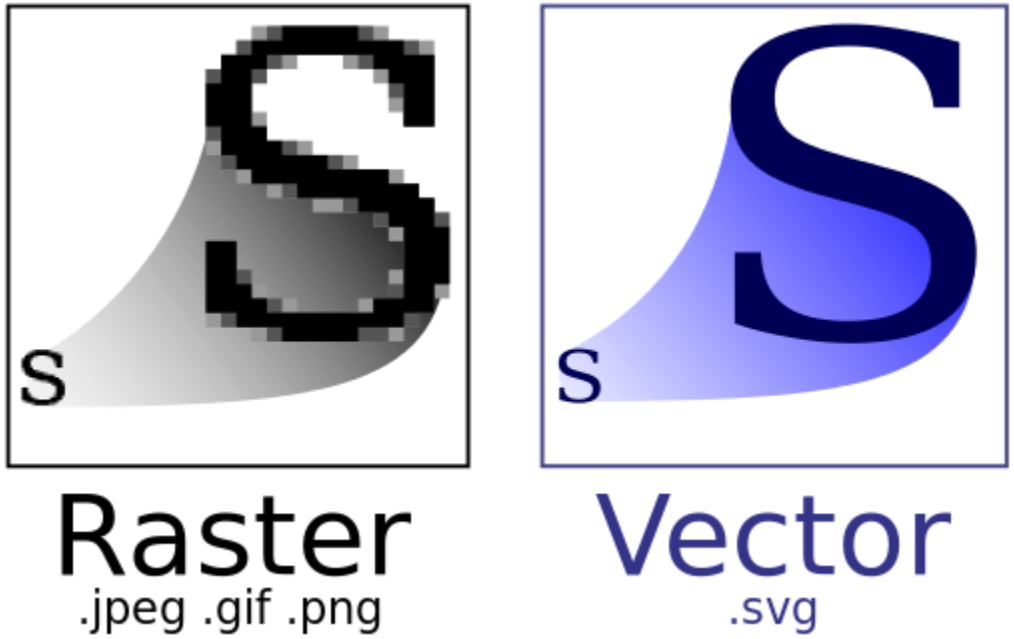


Image representation

Differences stemming from representation:

Vector images:

- Accurately represent geometric shapes at any scale
- Represent the algorithm to create the image,
  so rendering is done by repeating every calculation
- Compression is based on compressing the description and is lossless

Image representation

Differences stemming from representation:

Raster images:

- Approximately represent an image at one prescribed scale
- Represent the final result of a drawing algorithm,
  can be rendered without any complex operation
- Compression is based on compressing the result image and could be lossy

Image representation

Widely accepted file formats:

Vector:

- Postscript (.ps, .eps)
- Portable Data Format (.pdf) = zipped Postscript with some extras
- Scalable Vector Graphics (.svg) XML based language

Note: all of these support embedded raster images!

Image representation

Widely accepted file formats:

Raster:

- Portable Network Graphics (.png)
- Joint Photographic Experts Group (.jpeg, .jpg)
- Graphics Interchange Format (.gif)

Image representation

Under the hoods a Postscript file:

```
newpath
100 200 moveto
200 250 lineto
2 setlinewidth
stroke
showpage
```

Image representation

Under the hoods a Postscript file:

```
newpath
100 200 moveto
200 250 lineto
2 setlinewidth
stroke
showpage
```

Begin a new shape

Image representation

Under the hoods a Postscript file:

```
newpath
100 200 moveto
200 250 lineto
2 setlinewidth
stroke
showpage
```

Move the cursor to the position
x=100, y = 200

Image representation

Under the hoods a Postscript file:

```
newpath
100 200 moveto
200 250 lineto
2 setlinewidth
stroke
showpage
```

Record a line from the actual cursor to the point x=200, y = 250

Image representation

Under the hoods a Postscript file:

```
newpath
100 200 moveto
200 250 lineto
2 setlinewidth
stroke
showpage
```

Set the line width to 2

Image representation

Under the hoods a Postscript file:

```
newpath
100 200 moveto
200 250 lineto
2 setlinewidth
stroke
showpage
```

Finish the path

Image representation

Under the hoods a Postscript file:

```
newpath
100 200 moveto
200 250 lineto
2 setlinewidth
stroke
showpage
```

Finish the image

Image representation

Same image in svg:

```
<svg
    xmlns="http://www.w3.org/2000/svg"
    viewBox="0 0 743.75 1052.5"
    height="1052.5"
    width="743.75"
    version="1.1">
    <path
          style="stroke:#000000; stroke-width:20;"
          d="m 100,200 100,50" />
</svg>
```

Image representation

Same image in svg:

```
<svg
   xmlns="http://www.w3.org/2000/svg"
   viewBox="0 0 743.75 1052.5"
   height="1052.5"
   width="743.75"
   version="1.1">
   <path
         style="stroke:#000000; stroke-width:20;"
         d="m 100,200 100,50" />
</svg>
```

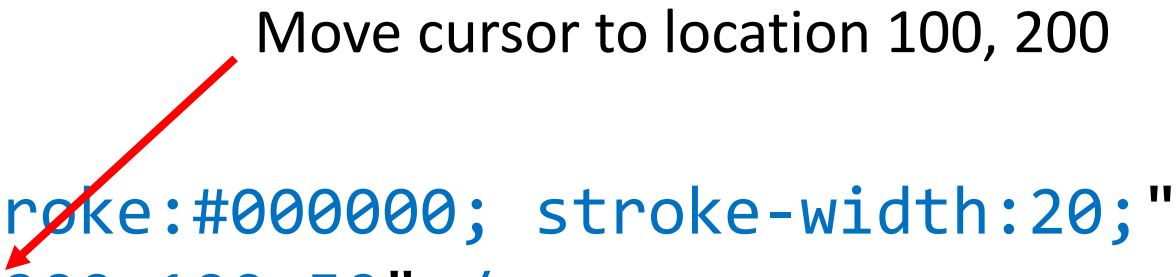


Image representation

Same image in svg:

```
<svg
    xmlns="http://www.w3.org/2000/svg"
    viewBox="0 0 743.75 1052.5"
    height="1052.5"
    width="743.75"
    version="1.1">
    <path
          style="stroke:#000000; stroke-width:20;"
          d="m 100,200 100,50" />
</svg>
```

Draw line to current location + (100, 50) units

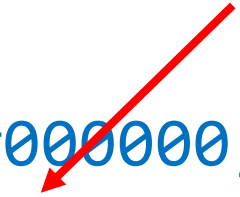


Image representation

Raster image representations:

A very simple raster format (like bitmap) just stores the

- width, height of the image,
- the number of bits to represent the colors
- and a long array of colors for each pixel

```
0000FF 0000FF 0000FF 0000FF 0000FF 0000FF 0000FF 0000FF
00FF00 FF0000 FFFFFF FF0000 FFFFFF FF0000 FFFFFF 00FF00
00FF00 FFFFFF FF0000 FFFFFF FF0000 FFFFFF FF0000 00FF00
00FF00 FF0000 FFFFFF FF0000 FFFFFF FF0000 FFFFFF 00FF00
00FF00 FFFFFF FF0000 FFFFFF FF0000 FFFFFF FF0000 00FF00
00FF00 FF0000 FFFFFF FF0000 FFFFFF FF0000 FFFFFF 00FF00
00FF00 FFFFFF FF0000 FFFFFF FF0000 FFFFFF FF0000 00FF00
0000FF 0000FF 0000FF 0000FF 0000FF 0000FF 0000FF 0000FF
```

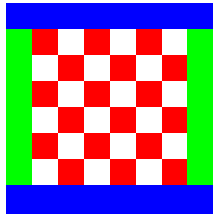
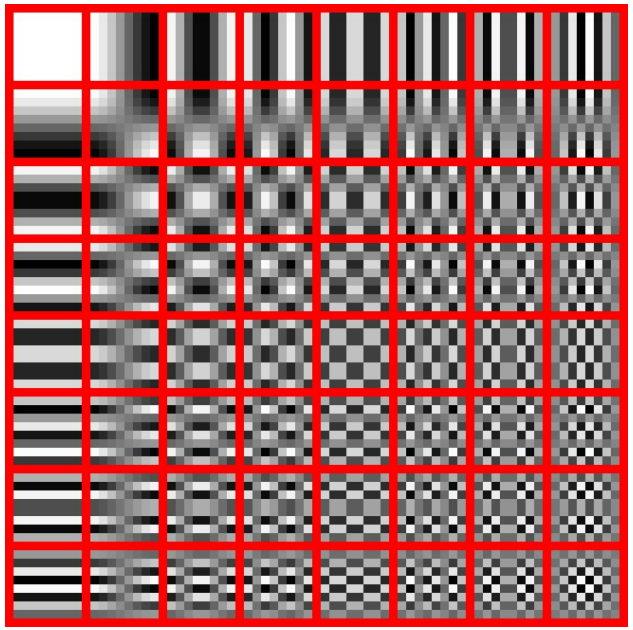


Image representation



Raster image representations:

JPEG does a

- Color space change (RGB->YCbCr)
- A Discrete Cosine Transform (on usually 8x8 blocks),
- Truncates the basis expansion coefficients
- Further compresses the resulting block with a generic lossless algorithm

Image representation

Raster image representations:

The basis truncation in the DCT phase of JPEG can create ringing artifacts on sharp edges:

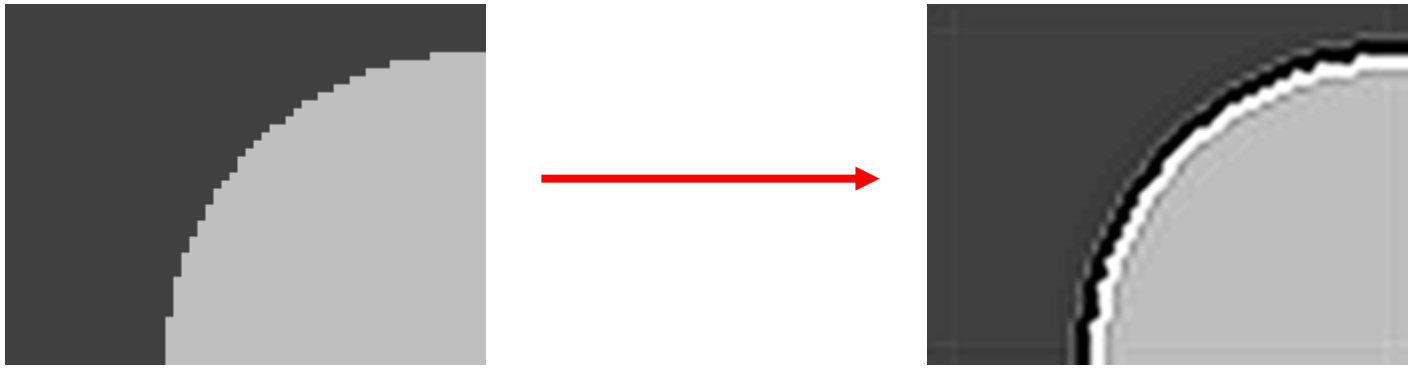


Image representation

Raster image representations:

PNG uses a lossless compression based on the public DEFLATE algorithm and a Difference encoding pre-compression sweep

PNG can more precisely represent sharp transitions and can compress non-photographic images (icons, text, line-art) much better than JPEG.

PNG also offers a fine-grained control over color and transparency representation.

Vector image primitives

Vector image primitives

The following building blocks luckily coincide:

- The primitives that one intuitionally expects to use for building up a drawing, plot, etc.
- The fundamental operations supported by vector graphics file formats
- The basic operations provided by drawing APIs and libraries.

Vector image primitives

Fundamental building blocks:

1D primitives:

- Straight lines
- Arcs (full, or partial circles, ellipses)
- Quadratic or Cubic Bezier curves

- Line styles (stroke)
- Fill styles

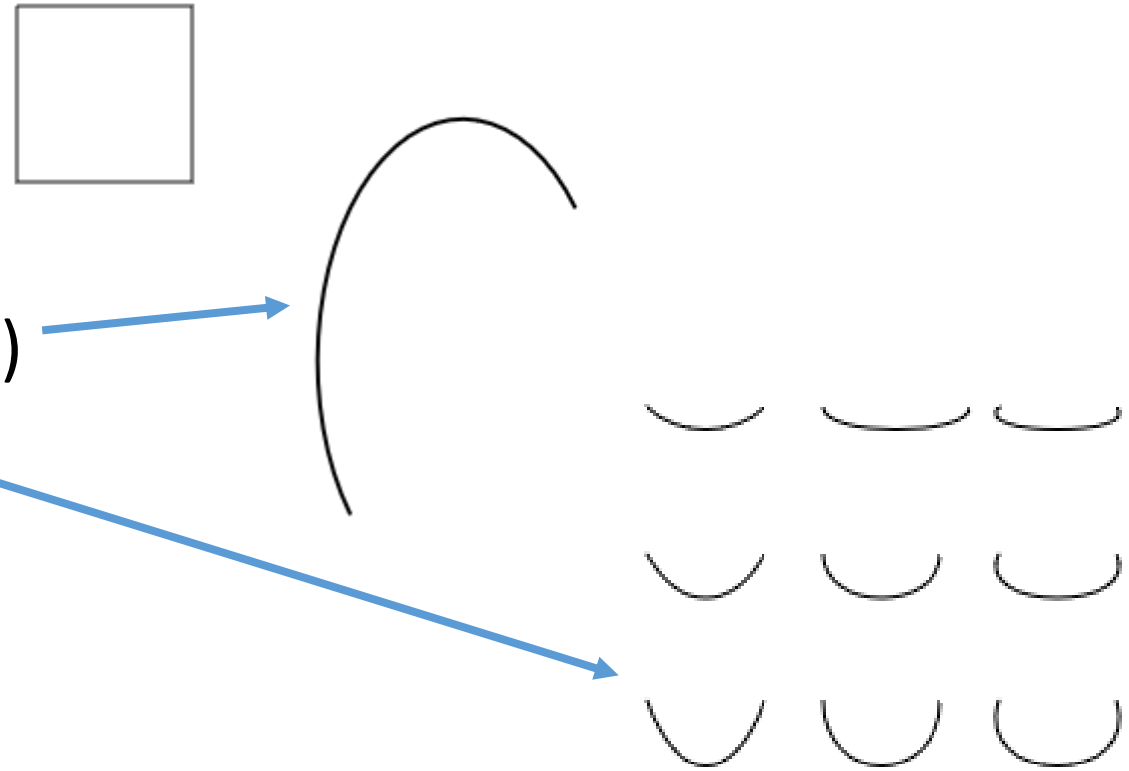
- Text

Vector image primitives

Fundamental building blocks:

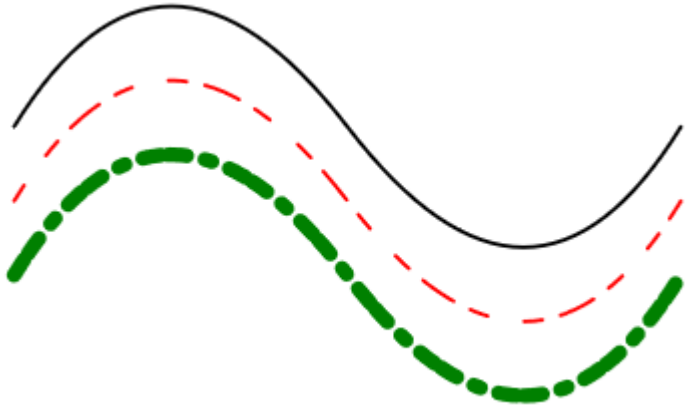
1D primitives:

- Straight lines
- Arcs (full, or partial circles, ellipses)
- Quadratic or Cubic Bezier curves



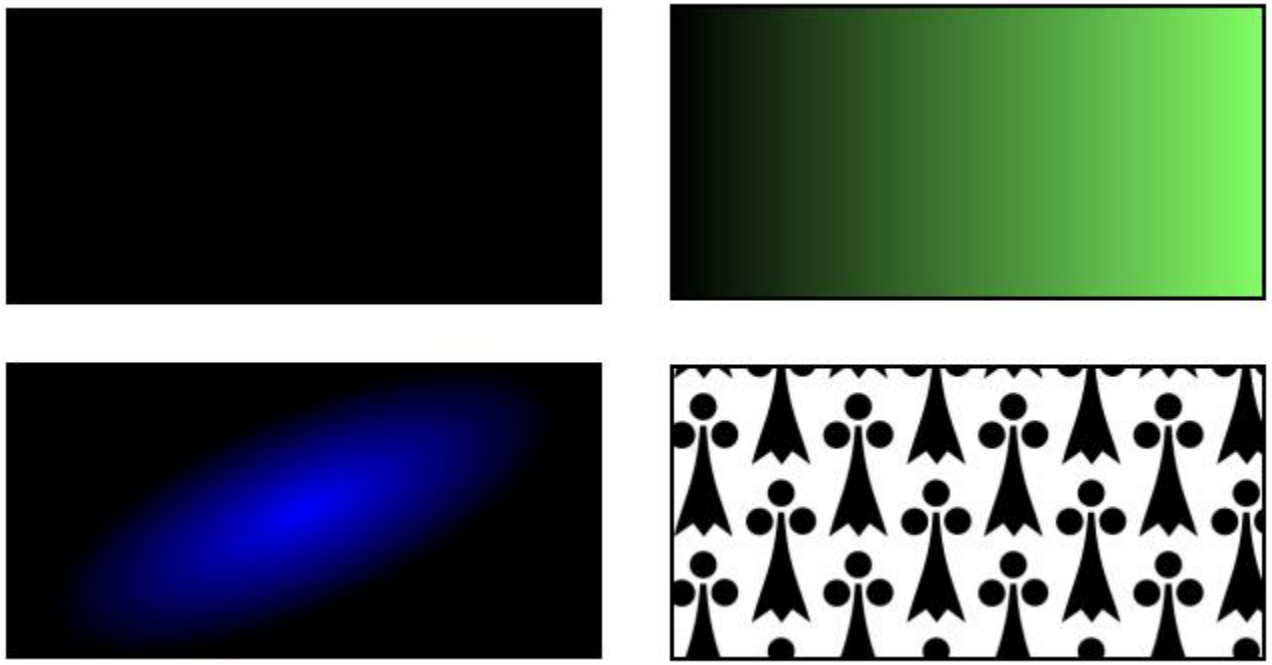
Vector image primitives

Fundamental building blocks:



Styles:

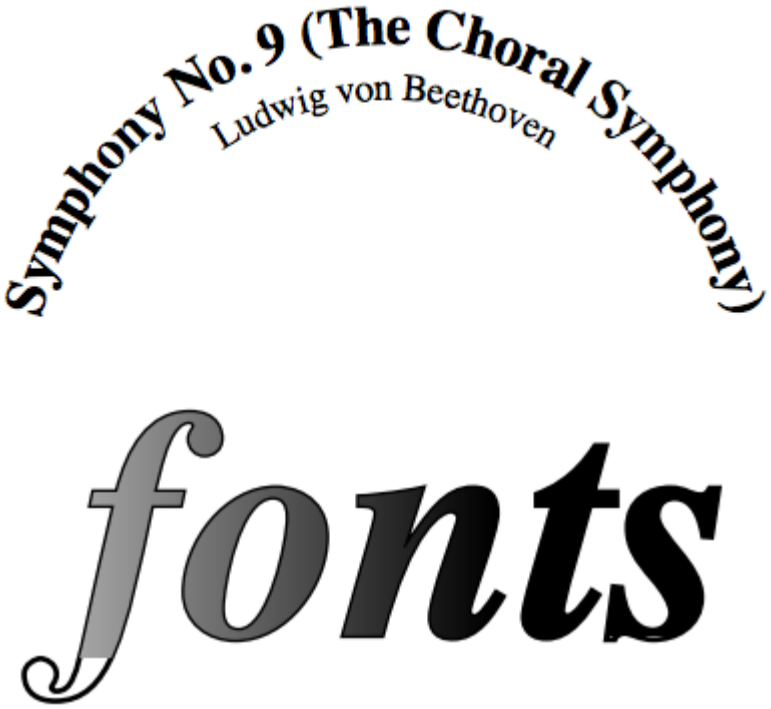
- Line styles (stroke)
- Fill styles



Vector image primitives

Fundamental building blocks:

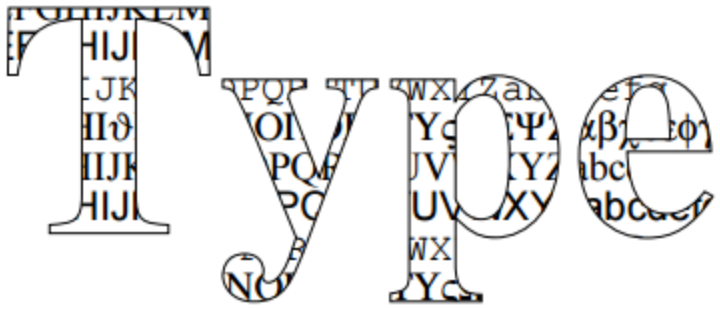
- Text rendering
  - Potentially following complex paths
  - Or having separate control over characters

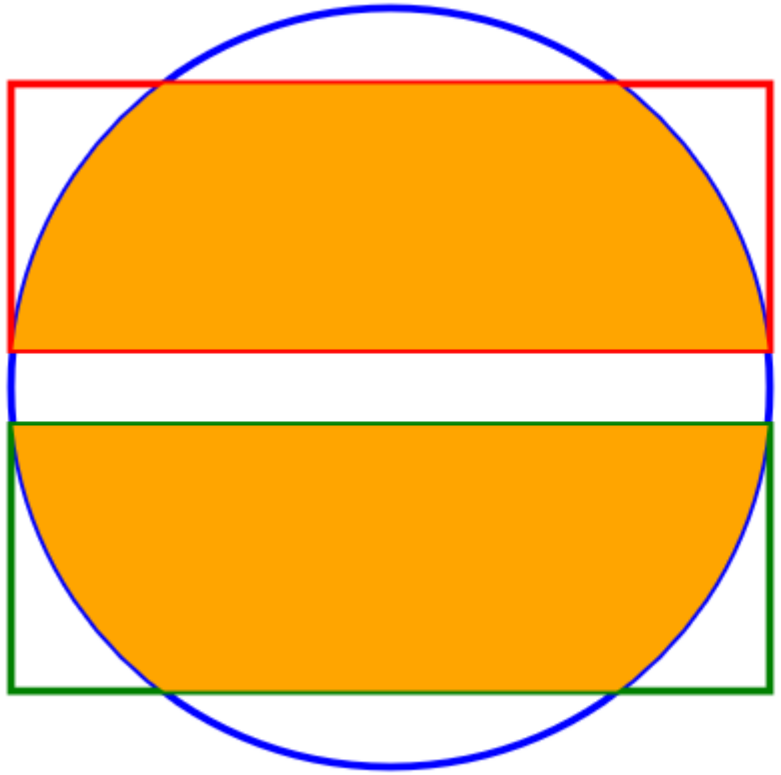


Vector image primitives

Fundamental building blocks:

- Clipping composition of primitives





Vector imaging

While there is an uncountable number of libraries and APIs to create vector graphics, we'll take a look to Cairo.

The reasons are:

- Cairo is a long standing well-established library successfully providing drawing facilities for hundreds of projects
- It is proposed for standardization of 2D graphics in C++

Cairo basics

```
#include <cairo.h>
#include <cairo/cairo-pdf.h>

int main()
{
  cairo_surface_t*  surface;
  surface = cairo_pdf_surface_create("image.pdf", 640, 480);
  cairo_t*          cr;
  cr = cairo_create(surface);
}
```

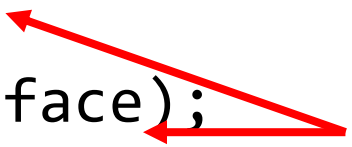
Cairo basics

```
#include <cairo.h>
#include <cairo/cairo-pdf.h>

int main()
{
  cairo_surface_t*  surface;
  surface = cairo_pdf_surface_create("image.pdf", 640, 480);
  cairo_t*          cr;
  cr = cairo_create(surface);
}
```

Surface represents the type of image we are creating, here a pdf file.

Cairo basics

```
#include <cairo.h>
#include <cairo/cairo-pdf.h>

int main()
{
  cairo_surface_t*  surface;
  surface = cairo_pdf_surface_create("image.pdf", 640, 480);
  cairo_t*          cr;
  cr = cairo_create(surface);
}
```



cr is the context representing the current drawing operations on the image.

Cairo basics

Extending the initialization with drawing a red square:

```
{
    cairo_set_line_width (cr, 10.0);
    cairo_set_source_rgba (cr, 1, 0, 0, 1);
    cairo_move_to(cr, 100, 100);
    cairo_line_to(cr, 200, 100);
    cairo_line_to(cr, 200, 200);
    cairo_line_to(cr, 100, 200);
    cairo_line_to(cr, 100, 100);
    cairo_stroke (cr);
}
```

Cairo basics

Extending the initialization with drawing a red square:

Set the current line width in the context

```
{
    cairo_set_line_width (cr, 10.0);
    cairo_set_source_rgba (cr, 1, 0, 0, 1);
    cairo_move_to(cr, 100, 100);
    cairo_line_to(cr, 200, 100);
    cairo_line_to(cr, 200, 200);
    cairo_line_to(cr, 100, 200);
    cairo_line_to(cr, 100, 100);
    cairo_stroke (cr);
}
```

Cairo basics

Extending the initialization with drawing a red square:

```
{
    cairo_set_line_width (cr, 10.0);
    cairo_set_source_rgba (cr, 1, 0, 0, 1);
    cairo_move_to(cr, 100, 100);
    cairo_line_to(cr, 200, 100);
    cairo_line_to(cr, 200, 200);
    cairo_line_to(cr, 100, 200);
    cairo_line_to(cr, 100, 100);
    cairo_stroke (cr);
}
```

Set the current color in the context (red, green, blue, alpha)

Cairo basics

Extending the initialization with drawing a red square:

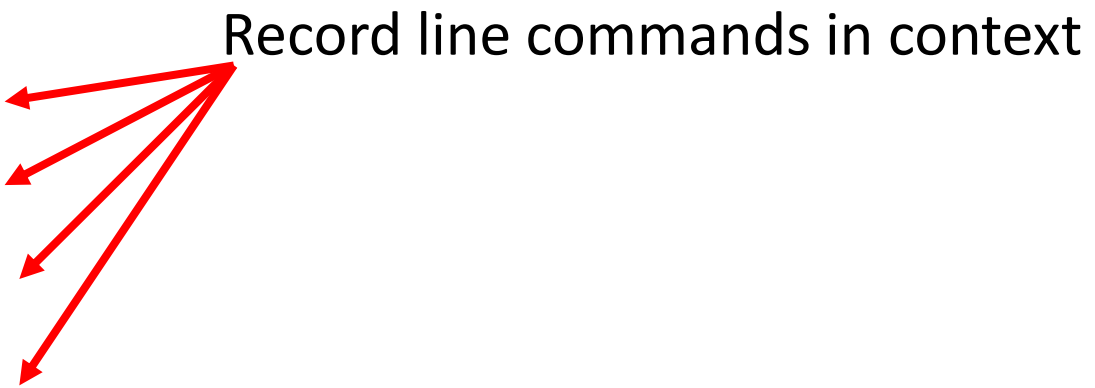
```
{
    cairo_set_line_width (cr, 10.0);
    cairo_set_source_rgba (cr, 1, 0, 0, 1);
    cairo_move_to(cr, 100, 100);
    cairo_line_to(cr, 200, 100);
    cairo_line_to(cr, 200, 200);
    cairo_line_to(cr, 100, 200);
    cairo_line_to(cr, 100, 100);
    cairo_stroke (cr);
}
```

Move context's cursor to location

Cairo basics

Extending the initialization with drawing a red square:

```
{
    cairo_set_line_width (cr, 10.0);
    cairo_set_source_rgba (cr, 1, 0, 0, 1);
    cairo_move_to(cr, 100, 100);
    cairo_line_to(cr, 200, 100);
    cairo_line_to(cr, 200, 200);
    cairo_line_to(cr, 100, 200);
    cairo_line_to(cr, 100, 100);
    cairo_stroke (cr);
}
```



Cairo basics

Extending the initialization with drawing a red square:

```
{
    cairo_set_line_width (cr, 10.0);
    cairo_set_source_rgba (cr, 1, 0, 0, 1);
    cairo_move_to(cr, 100, 100);
    cairo_line_to(cr, 200, 100);
    cairo_line_to(cr, 200, 200);
    cairo_line_to(cr, 100, 200);
    cairo_line_to(cr, 100, 100);
    cairo_stroke (cr);
}
```

Finish shape! The state of the context determines all the properties of the shape at this point

Cairo basics

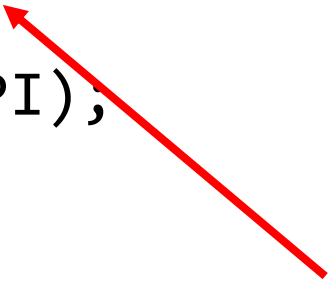
Now draw a blue filled circle:

```
{
    cairo_set_source_rgba (cr, 0, 0, 1, 1);
    cairo_arc (cr, 300, 100, 90.0, 0, 2*M_PI);
    cairo_fill (cr);
}
```

Cairo basics

Now draw a blue filled circle:

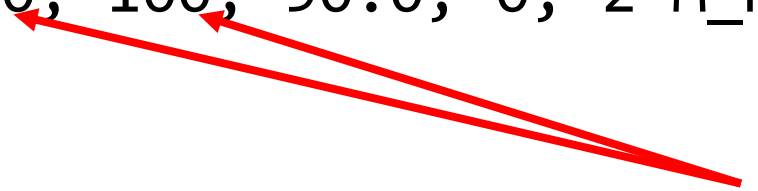
```
{
    cairo_set_source_rgba (cr, 0, 0, 1, 1);
    cairo_arc (cr, 300, 100, 90.0, 0, 2*M_PI);
    cairo_fill (cr);
}
```



Cairo basics

Now draw a blue filled circle:

```
{
    cairo_set_source_rgba (cr, 0, 0, 1, 1);
    cairo_arc (cr, 300, 100, 90.0, 0, 2*M_PI);
    cairo_fill (cr);
}
```

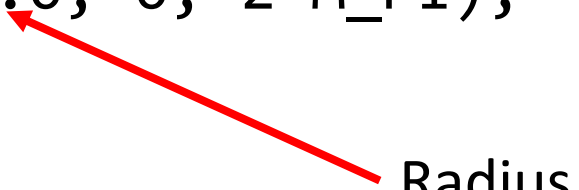


Arc center position (x, y)

Cairo basics

Now draw a blue filled circle:

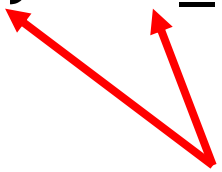
```
{
    cairo_set_source_rgba (cr, 0, 0, 1, 1);
    cairo_arc (cr, 300, 100, 90.0, 0, 2*M_PI);
    cairo_fill (cr);
}
```



Cairo basics

Now draw a blue filled circle:

```
{
    cairo_set_source_rgba (cr, 0, 0, 1, 1);
    cairo_arc (cr, 300, 100, 90.0, 0, 2*M_PI);
    cairo_fill (cr);
}
```



Start and end angle in radians

Cairo basics

Now draw a blue filled circle:

```
{
    cairo_set_source_rgba (cr, 0, 0, 1, 1);
    cairo_arc (cr, 300, 100, 90.0, 0, 2*M_PI);
    cairo_fill (cr);
}
```

Finish shape, this time with a fill (no outline is drawn)

Cairo basics

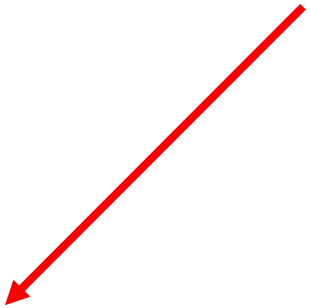
Very similarly we can write text to the context:

```
{
   cairo_set_source_rgb(cr, 0, 0, 0);
   cairo_select_font_face(cr, "Sans",
                          CAIRO_FONT_SLANT_NORMAL,
                          CAIRO_FONT_WEIGHT_NORMAL);
   cairo_set_font_size(cr, 40.0);
   cairo_move_to(cr, 10.0, 300.0);
   cairo_show_text(cr, "Hello Cairo!");
}
```

Cairo basics

Very similarly we can write text to the context:

Set active font for the context



```
{
    cairo_set_source_rgb(cr, 0, 0, 0);
    cairo_select_font_face(cr, "Sans",
                           CAIRO_FONT_SLANT_NORMAL,
                           CAIRO_FONT_WEIGHT_NORMAL);
    cairo_set_font_size(cr, 40.0);
    cairo_move_to(cr, 10.0, 300.0);
    cairo_show_text(cr, "Hello Cairo!");
}
```

Cairo basics

Very similarly we can write text to the context:

```
{
    cairo_set_source_rgb(cr, 0, 0, 0);
    cairo_select_font_face(cr, "Sans",
                           CAIRO_FONT_SLANT_NORMAL,
                           CAIRO_FONT_WEIGHT_NORMAL);
    cairo_set_font_size(cr, 40.0);
    cairo_move_to(cr, 10.0, 300.0);
    cairo_show_text(cr, "Hello Cairo!");
}
```

Set font size for the context

Cairo basics

Very similarly we can write text to the context:

```
{
    cairo_set_source_rgb(cr, 0, 0, 0);
    cairo_select_font_face(cr, "Sans",
                           CAIRO_FONT_SLANT_NORMAL,
                           CAIRO_FONT_WEIGHT_NORMAL);
    cairo_set_font_size(cr, 40.0);
    cairo_move_to(cr, 10.0, 300.0);
    cairo_show_text(cr, "Hello Cairo!");
}
```

Set cursor (top left point of text)

Cairo basics

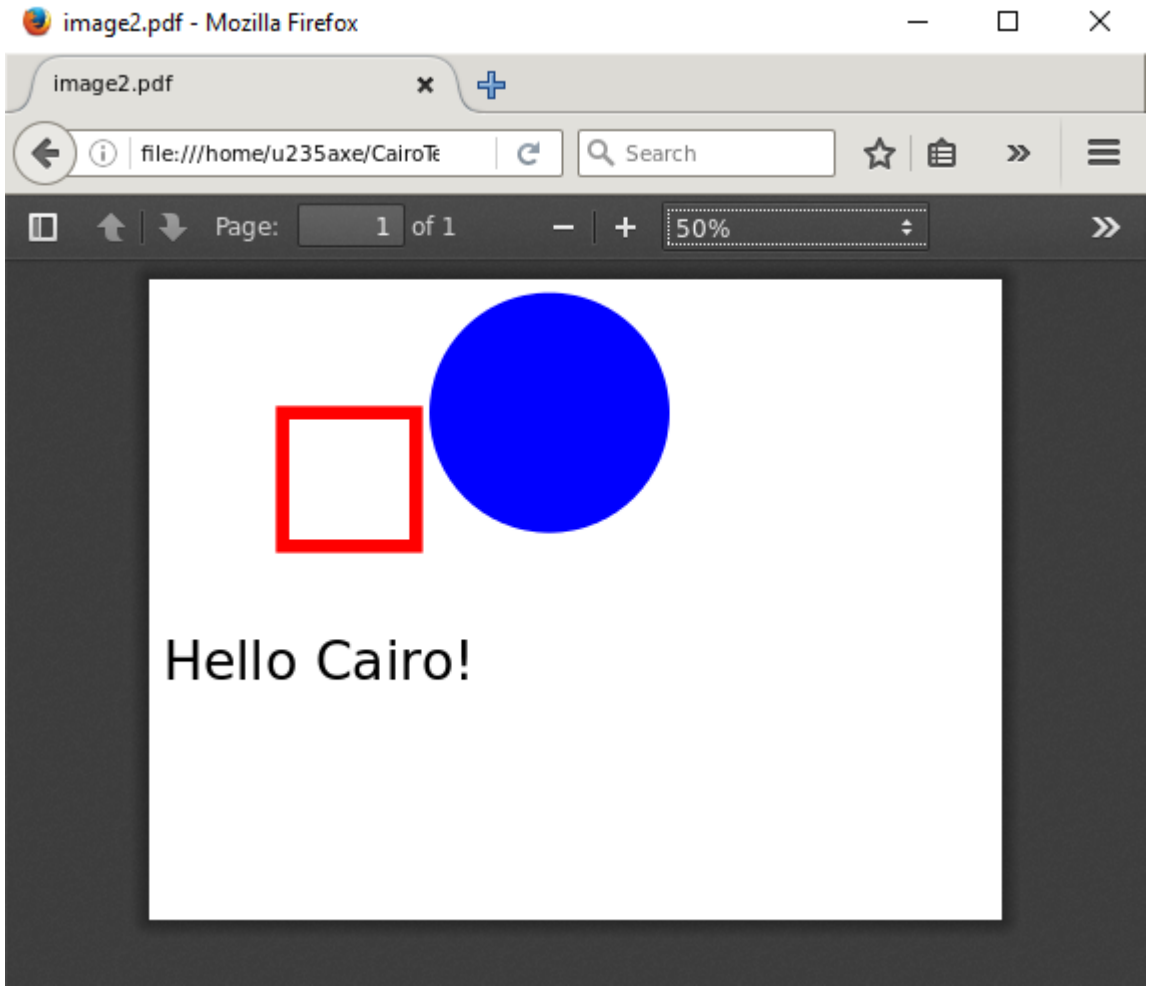
Very similarly we can write text to the context:

```
{
    cairo_set_source_rgb(cr, 0, 0, 0);
    cairo_select_font_face(cr, "Sans",
                           CAIRO_FONT_SLANT_NORMAL,
                           CAIRO_FONT_WEIGHT_NORMAL);
    cairo_set_font_size(cr, 40.0);
    cairo_move_to(cr, 10.0, 300.0);
    cairo_show_text(cr, "Hello Cairo!");
}
```

Save the text to the context with all the active properties above.

Cairo basics

Final result pdf looks like this:



Cairo basics

Observations:

- The very same commands are used, like in postscript or svg.
- The implicit content of the context may lead to surprises...
- The implicit mutable state endangers multithreading by construction when shared between multiple threads.

Drawing to a window

To have a functioning window, the following two things needed:

- Create the window
- Process the events that happen to the window

Drawing to a window

Under *nix the ancient X11 windowing system still dominant.

To set up a window:

```
Display* dsp = XOpenDisplay(NULL);
int screen = DefaultScreen(dsp);
Drawable da = XCreateSimpleWindow(dsp,
                                  DefaultRootWindow(dsp),
                                  0, 0, 640, 480, 0, 0, 0);
XSelectInput(dsp, da, KeyPressMask);
XMapWindow(dsp, da);
```

Drawing to a window

Under *nix the ancient X11 windowing system still dominant.

To set up a window:

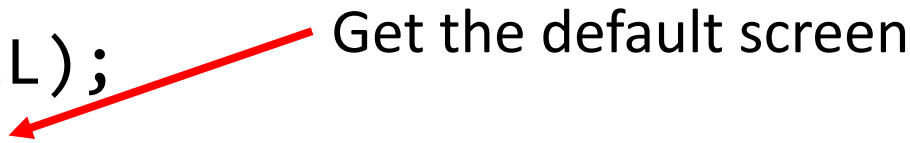

```
Display* dsp = XOpenDisplay(NULL);
int screen = DefaultScreen(dsp);
Drawable da = XCreateSimpleWindow(dsp,
                                  DefaultRootWindow(dsp),
                                  0, 0, 640, 480, 0, 0, 0);
XSelectInput(dsp, da, KeyPressMask);
XMapWindow(dsp, da);
```

Drawing to a window

Under *nix the ancient X11 windowing system still dominant.

To set up a window:

```
Display* dsp = XOpenDisplay(NULL);
int screen = DefaultScreen(dsp);
Drawable da = XCreateSimpleWindow(dsp,
                                  DefaultRootWindow(dsp),
                                  0, 0, 640, 480, 0, 0, 0);
XSelectInput(dsp, da, KeyPressMask);
XMapWindow(dsp, da);
```



Drawing to a window

Under *nix the ancient X11 windowing system still dominant.

To set up a window:

```
Display* dsp = XOpenDisplay(NULL);
int screen = DefaultScreen(dsp);
Drawable da = XCreateSimpleWindow(dsp,
                                  DefaultRootWindow(dsp),
                                  0, 0, 640, 480, 0, 0, 0);
XSelectInput(dsp, da, KeyPressMask);
XMapWindow(dsp, da);
```

Drawing to a window

Under *nix the ancient X11 windowing system still dominant.

To set up a window:

```
Display* dsp = XOpenDisplay(NULL);
int screen = DefaultScreen(dsp);
Drawable da = XCreateSimpleWindow(dsp,
                                  DefaultRootWindow(dsp),
                                  0, 0, 640, 480, 0, 0, 0);
XSelectInput(dsp, da, KeyPressMask);
XMapWindow(dsp, da);
```

Enable key press notifications for this window

Drawing to a window

Under *nix the ancient X11 windowing system still dominant.

To set up a window:

```
Display* dsp = XOpenDisplay(NULL);
int screen = DefaultScreen(dsp);
Drawable da = XCreateSimpleWindow(dsp,
                                  DefaultRootWindow(dsp),
                                  0, 0, 640, 480, 0, 0, 0);
XSelectInput(dsp, da, KeyPressMask);
XMapWindow(dsp, da);
```


Finish setup, show window

Drawing to a window

Set up a Cairo context for the window:

```
cairo_surface_t* surface =
    cairo_xlib_surface_create(dsp, da,
                              DefaultVisual(dsp, screen),
                              640, 480);

cairo_xlib_surface_set_size(surface, 640, 480);

cairo_t* cr = cairo_create(surface);
```

Drawing to a window

Set up a Cairo context for the window:

Associate the X window with a Cairo surface

```
cairo_surface_t* surface =
    cairo_xlib_surface_create(dsp, da,
                              DefaultVisual(dsp, screen),
                              640, 480);

cairo_xlib_surface_set_size(surface, 640, 480);

cairo_t* cr = cairo_create(surface);
```

Drawing to a window

Set up a Cairo context for the window:

Set the surface size
(should be called at every window resize!)

```
cairo_surface_t* surface =
    cairo_xlib_surface_create(dsp, da,
                              DefaultVisual(dsp, screen),
                              640, 480);

cairo_xlib_surface_set_size(surface, 640, 480);

cairo_t* cr = cairo_create(surface);
```

Drawing to a window

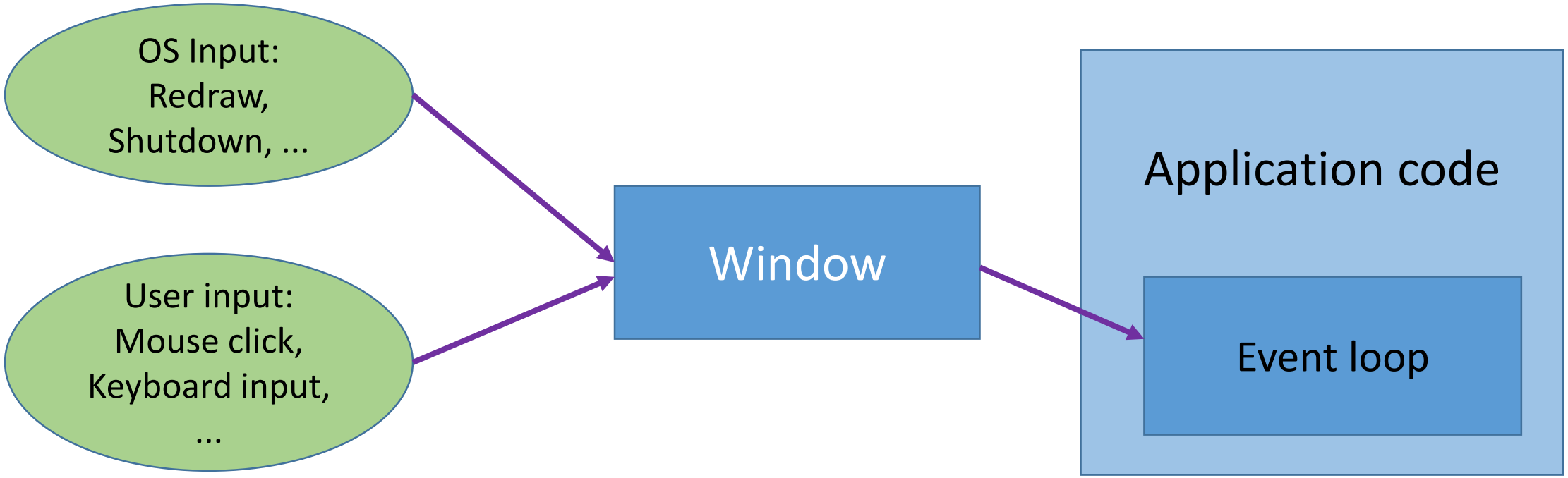
Set up a Cairo context for the window:

```
cairo_surface_t* surface =
    cairo_xlib_surface_create(dsp, da,
                              DefaultVisual(dsp, screen),
                              640, 480);

cairo_xlib_surface_set_size(surface, 640, 480);

cairo_t* cr = cairo_create(surface);
```

Create the Cairo rendering context

Drawing to a window

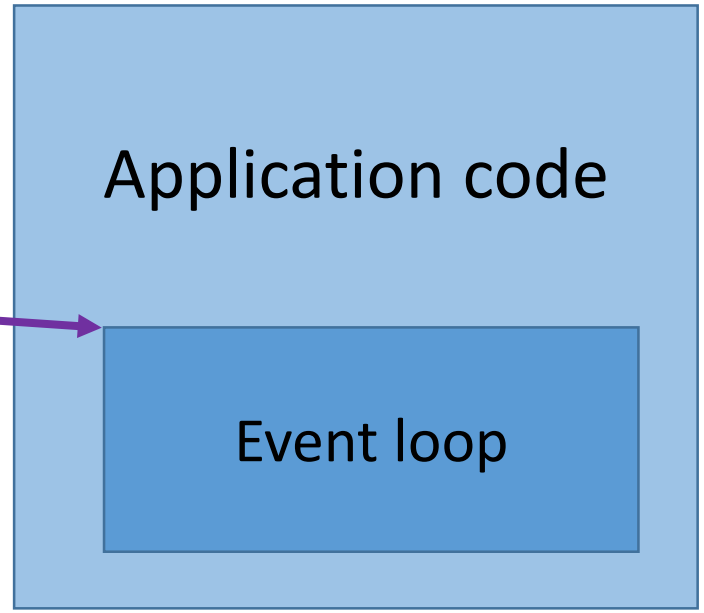
Windows are not just display devices, but they are the link of communication from the user and the OS to the application:



Drawing to a window

The event loop is an infinite loop processing the events of the window:

```
Display* dsp; XEvent e;
for(;;)
{
    if(XPending(dsp))
    {
       XNextEvent(dsp, &e);
    }
    if(e.type == KeyPress){ return 0; }
}
```



Drawing to a window

The event loop is an infinite loop processing the events of the window:

```
Display* dsp; XEvent e;
for(;;)
{
    if(XPending(dsp))
    {
       XNextEvent(dsp, &e);
    }
    if(e.type == KeyPress){ return 0; }
}
```

Infinite loop processing the events

Drawing to a window

The event loop is an infinite loop processing the events of the window:

```
Display* dsp; XEvent e;
for(;;)
{
    if(XPending(dsp))
    {
       XNextEvent(dsp, &e);
    }
    if(e.type == KeyPress){ return 0; }
}
```

If there is a pending event

Take it from the message queue and copy to the variable e.

Drawing to a window

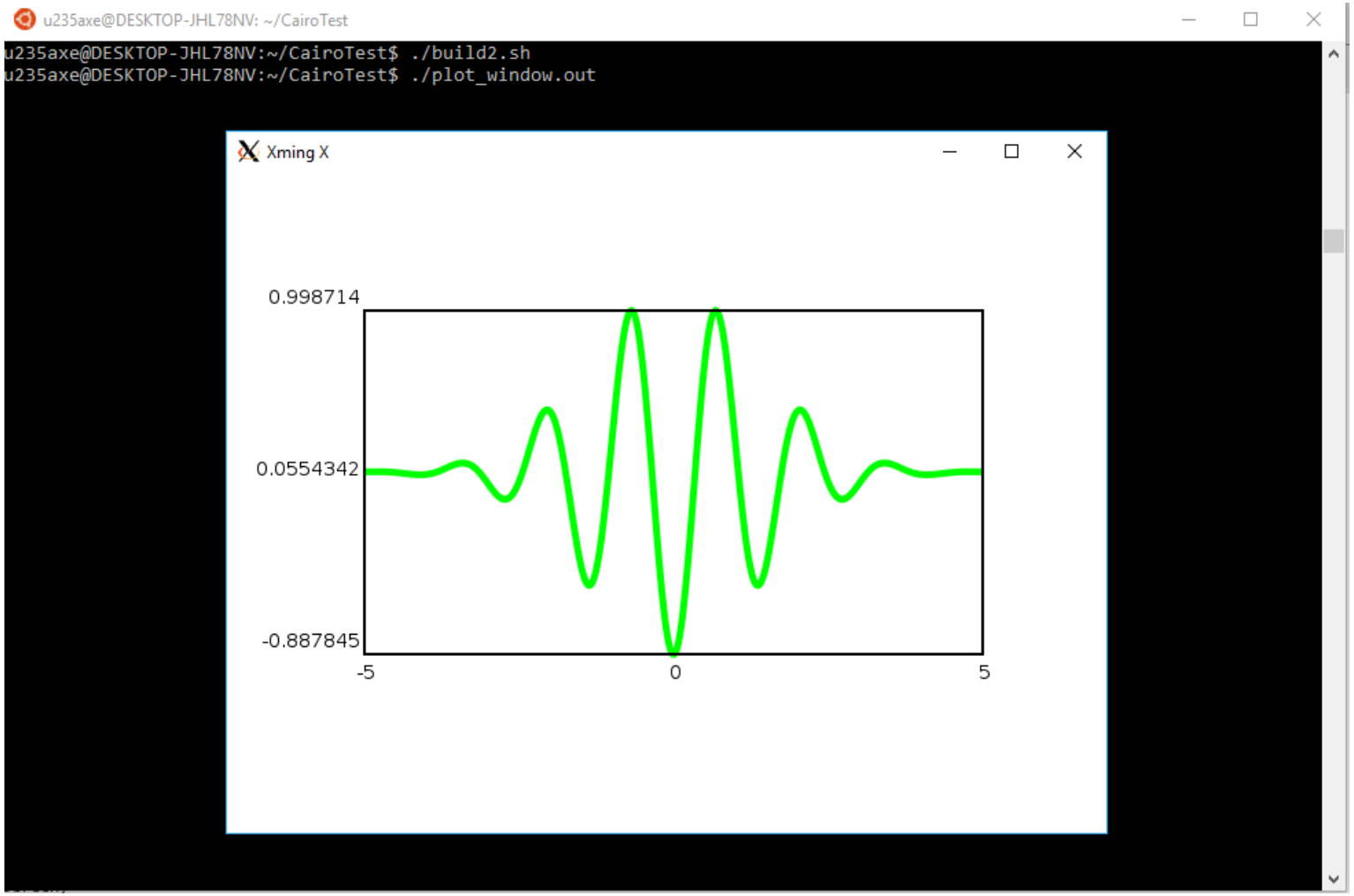
The event loop is an infinite loop processing the events of the window:

```
Display* dsp; XEvent e;
for(;;)
{
    if(XPending(dsp))
    {
       XNextEvent(dsp, &e);
    }
    if(e.type == KeyPress){ return 0; }
}
```

If the event was a key press on the keyboard exit the loop

Simple plotting sample

We created a simple demo plotting an arbitrary function with Cairo to an X11 window:



Going one step further…

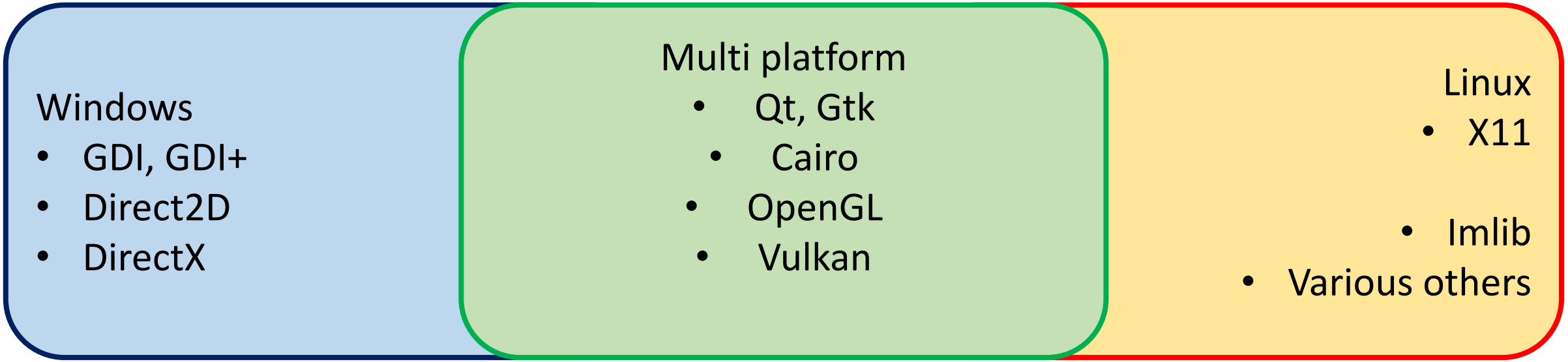
Going one step further…

There are compromises when choosing a rendering API:

- How versatile rendering is supported out of the box?
  - Different line styles, text rendering, shading, clipping, ...
- Is it hardware accelerated?
  - Can we plot a long dataset in a reasonable time?
- Does it support 3D rendering also?

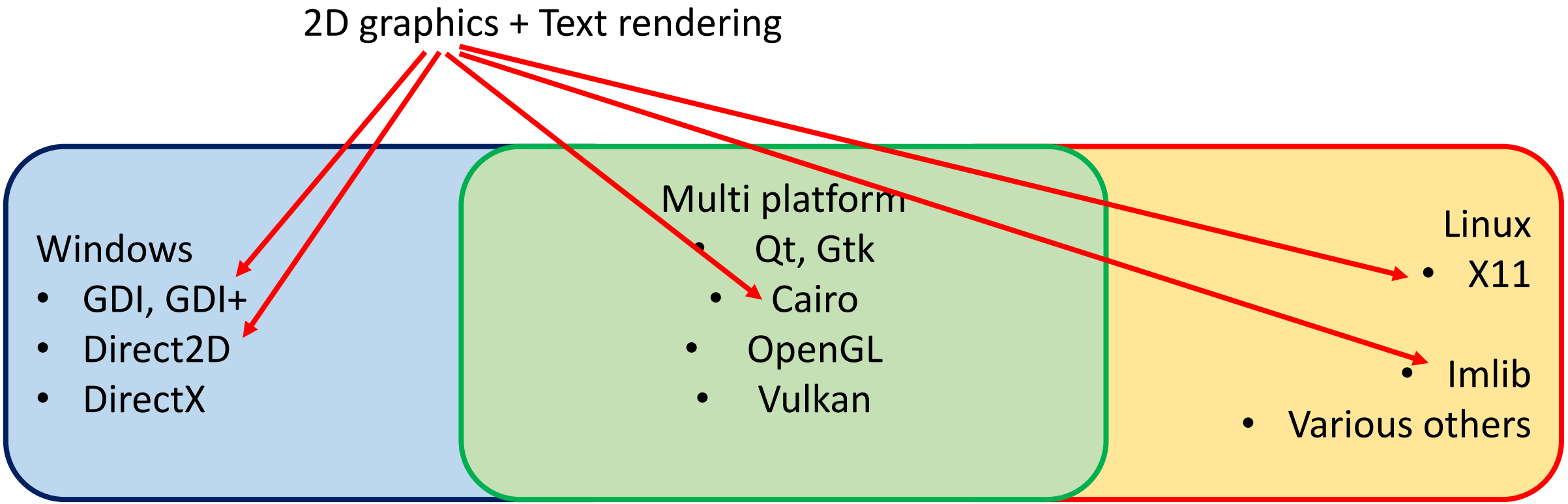
Graphics APIs

Current set of Graphics and related APIs:



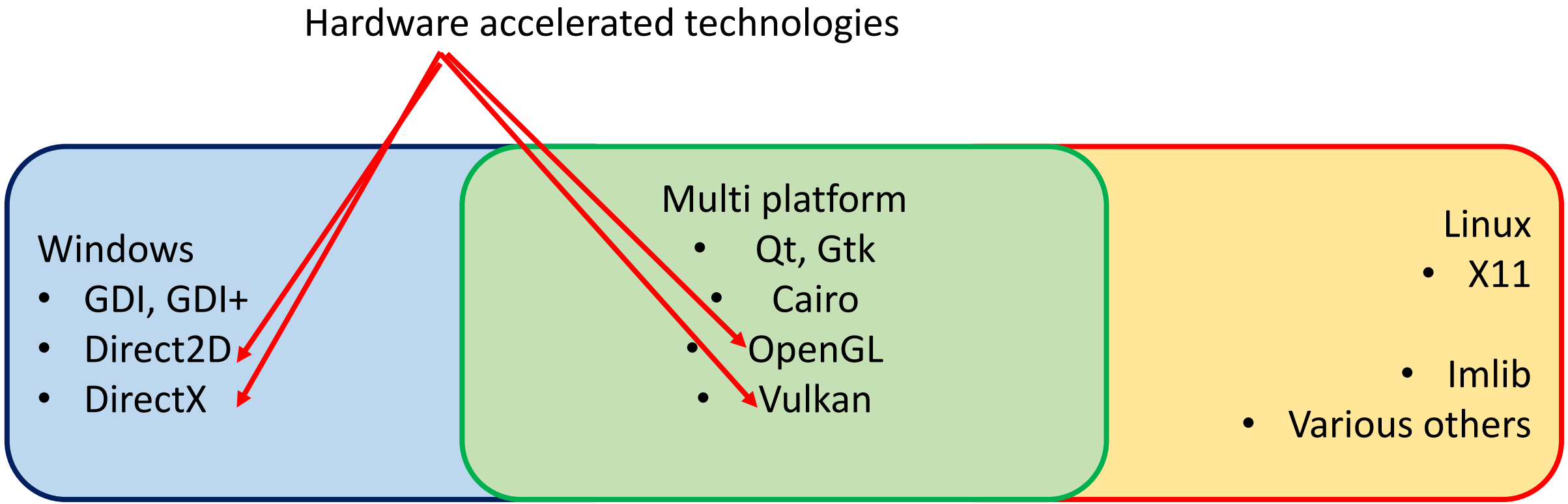
Graphics APIs

Current set of Graphics and related APIs:



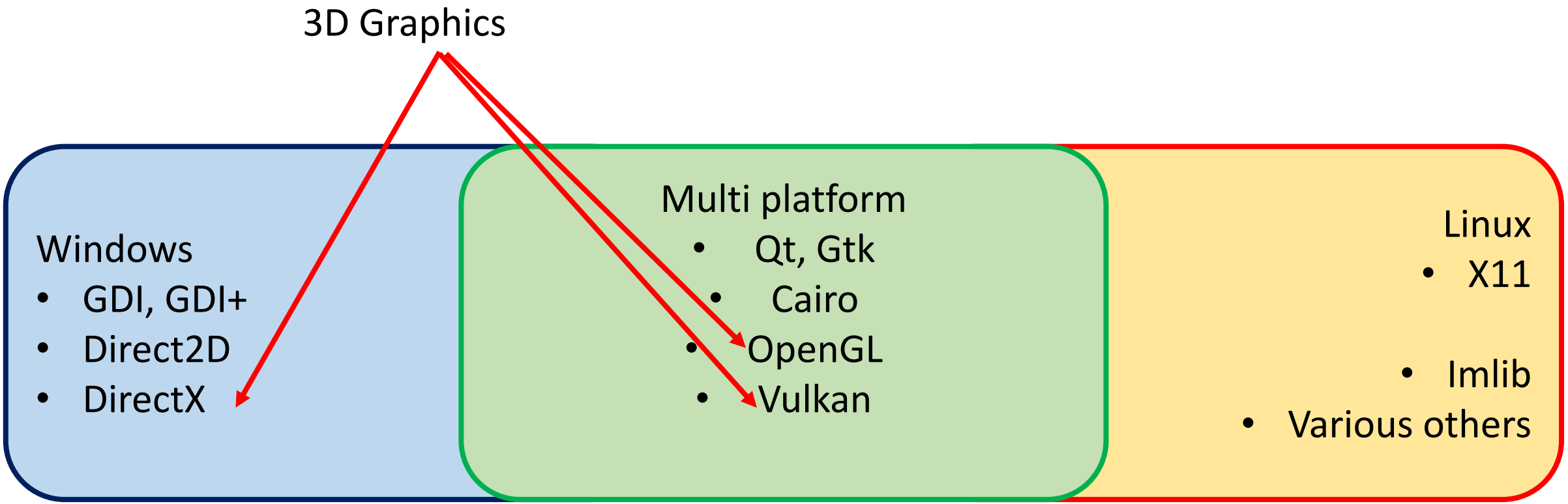
Graphics APIs

Current set of Graphics and related APIs:



Graphics APIs

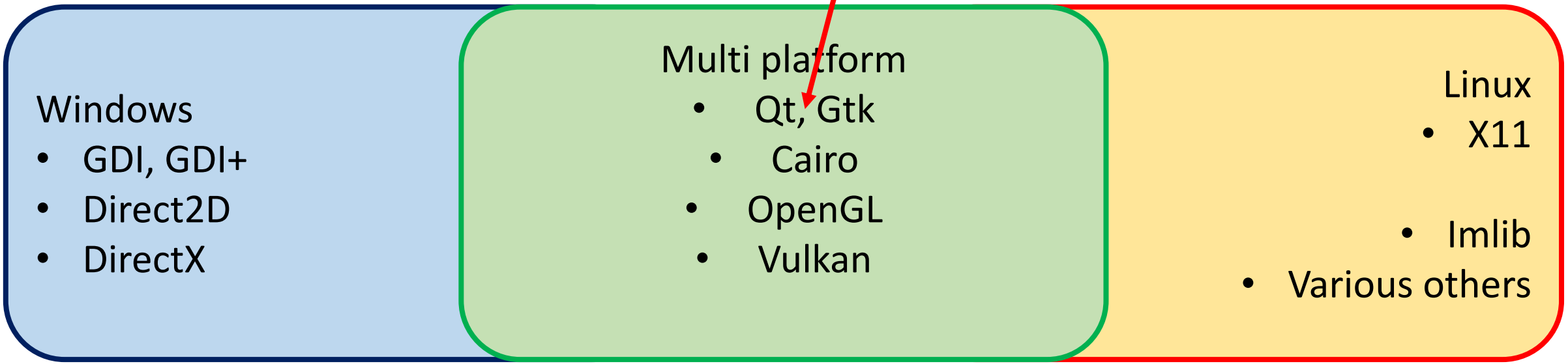
Current set of Graphics and related APIs:



Graphics APIs

Current set of Graphics and related APIs:

GUI toolkits
Qt offers a full development solution from graphics to networking but is an overkill for smaller tasks



Graphics APIs

Difference between 3D Hardware accelerated APIs and 2D graphics APIs:

- 3D APIs are tuned for performance, not so well for professional graphics

- It is harder to create nice looking curves and styles like dashing, etc..
  - Typically text rendering is not part of them!

- They massively outperform everything else in terms of speed
  - They are suitable for real-time rendering

Quick Into to OpenGL

Quick Into to OpenGL

- OpenGL is a well-established graphics standard
- Implemented and supported on all desktop and many mobile platforms (OpenGL ES)
- It is quite old and outdated, but simple to start and supports most sophisticated levels of accessing the GPU

Quick Into to OpenGL

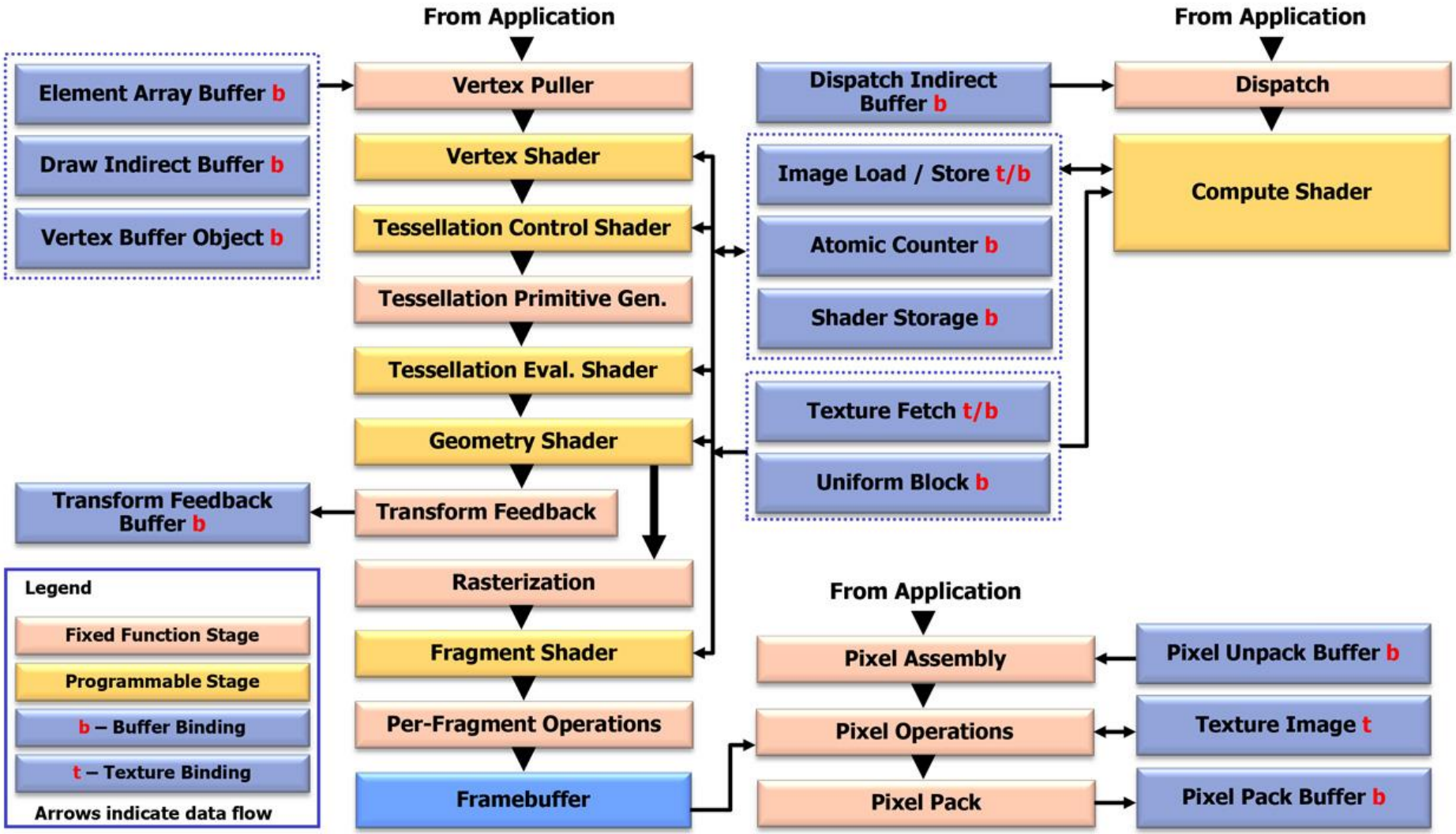
- OpenGL has versions ranging from 1.0 to 4.5
- Different platforms and vendors and drivers are at different versions
  - prepare for surprises...
- Can be very challenging to find proper sample codes and good practices for a certain version

Quick Into to OpenGL

- Now we'd like to introduce the more modern, programmable part of it

- The following codes are for OpenGL 4.3

OpenGL 4.3 with Compute Shaders

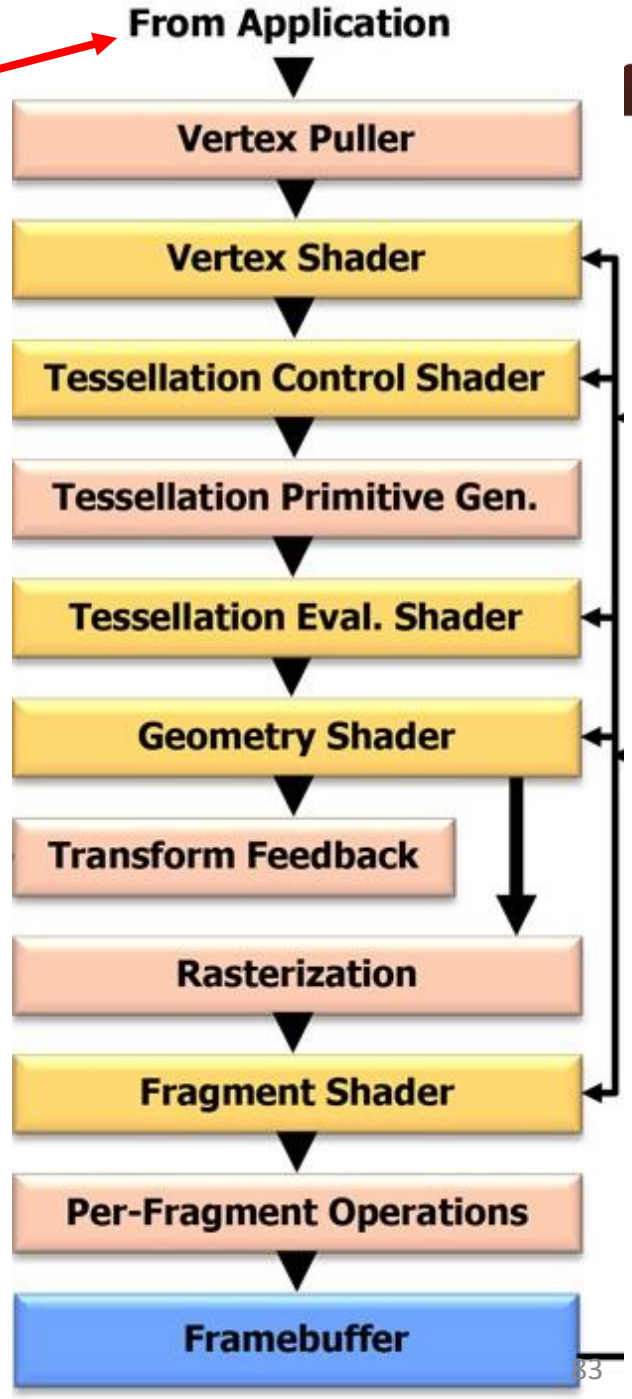


Programmable pipeline

Vertex data is created on the CPU (host) side and sent to the GPU

Vertex data means an array of multiple bunches of 1-4 components of floats, like:

```
struct Vertex
{
	float3 position;
	float3 normal;
	float4 color;
};
std::vector<Vertex> vertices(100000);
```

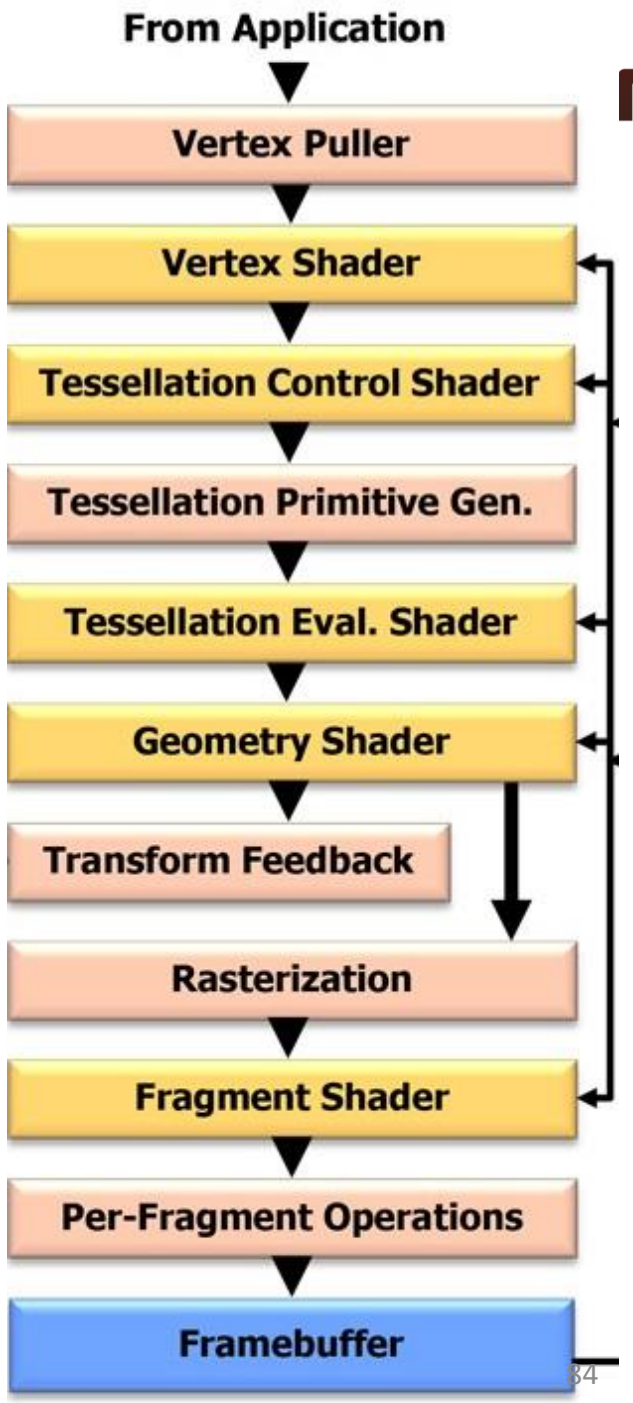


Programmable pipeline

Vertex Puller (Input Assembler):

The buffers and associated datas are collected and set to the pipeline

Primitives (lines, triangles) are formed and prepared for vertex shader to operate on

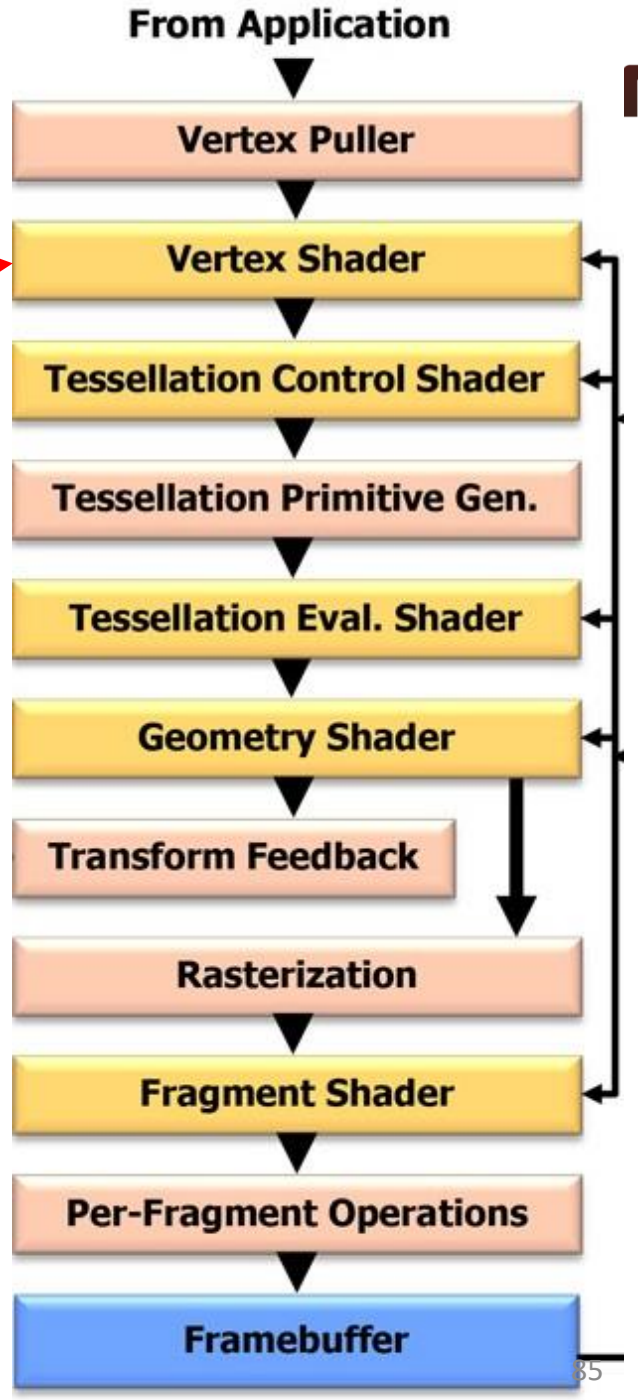


Programmable pipeline

Vertex Shader:

This programmable stage performs some arbitrary operation on the input vertex data

Typical usage: geometric transformations

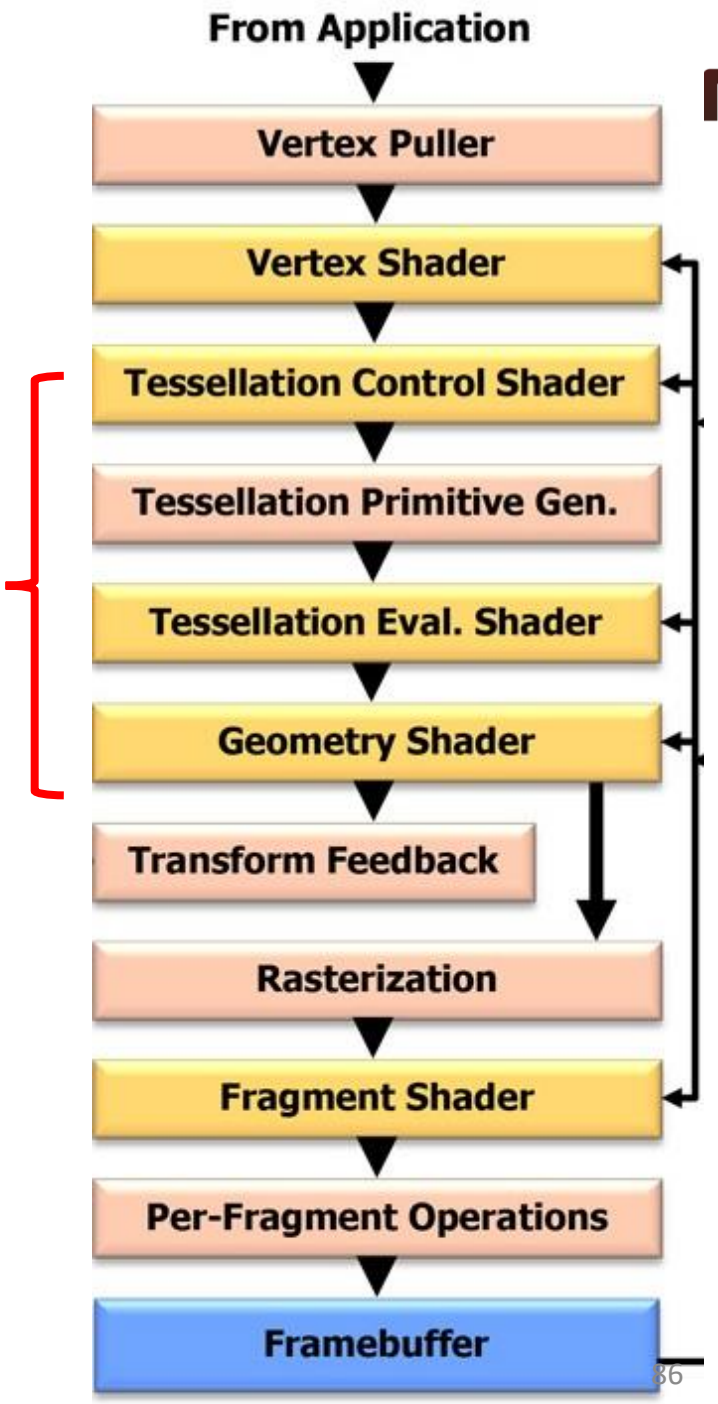


Programmable pipeline

Tessellation and Geometry shader:

These stages cover cases where new vertices can be produced programmatically under some constraints

Typically used for adaptive subdivision and interpolation



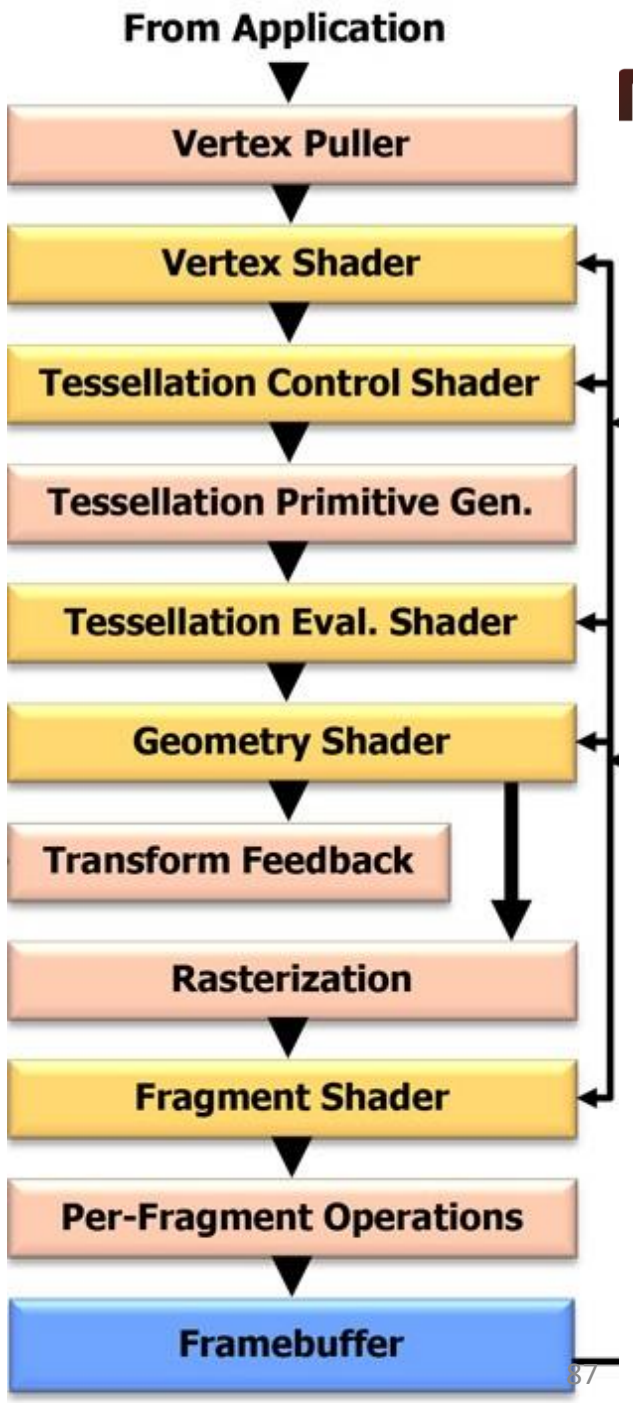
Programmable pipeline

Feedback or Stream output:

The so far processed vertices can be read back to a temporary buffer on the GPU without any pixel rendering

Useful for some complex multi-pass algorithms

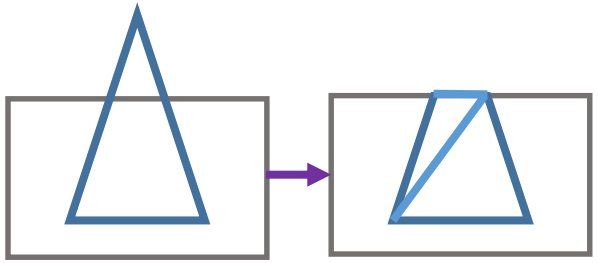
These buffers later can be set again as pipeline inputs at the top



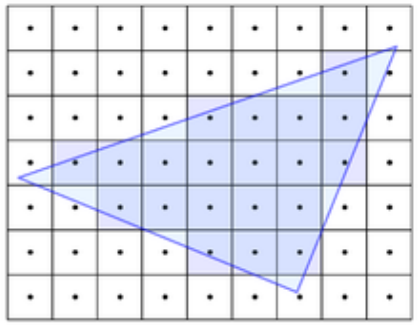
Programmable pipeline

Rasterization:
In this non-programmable stage:

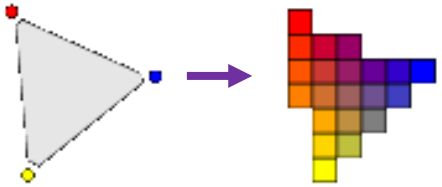
- Vertices are clipped

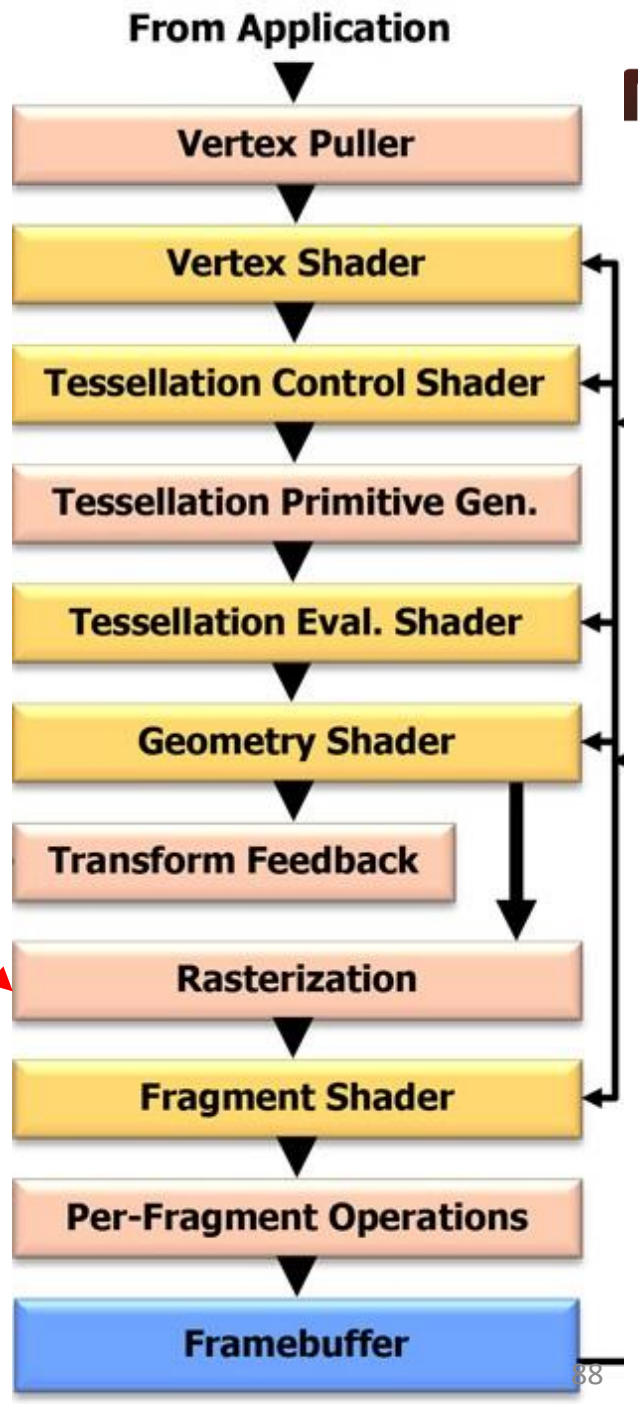


- The vector representation converted to a pixel based approximation



- Vertex components are interpolated over the primitives



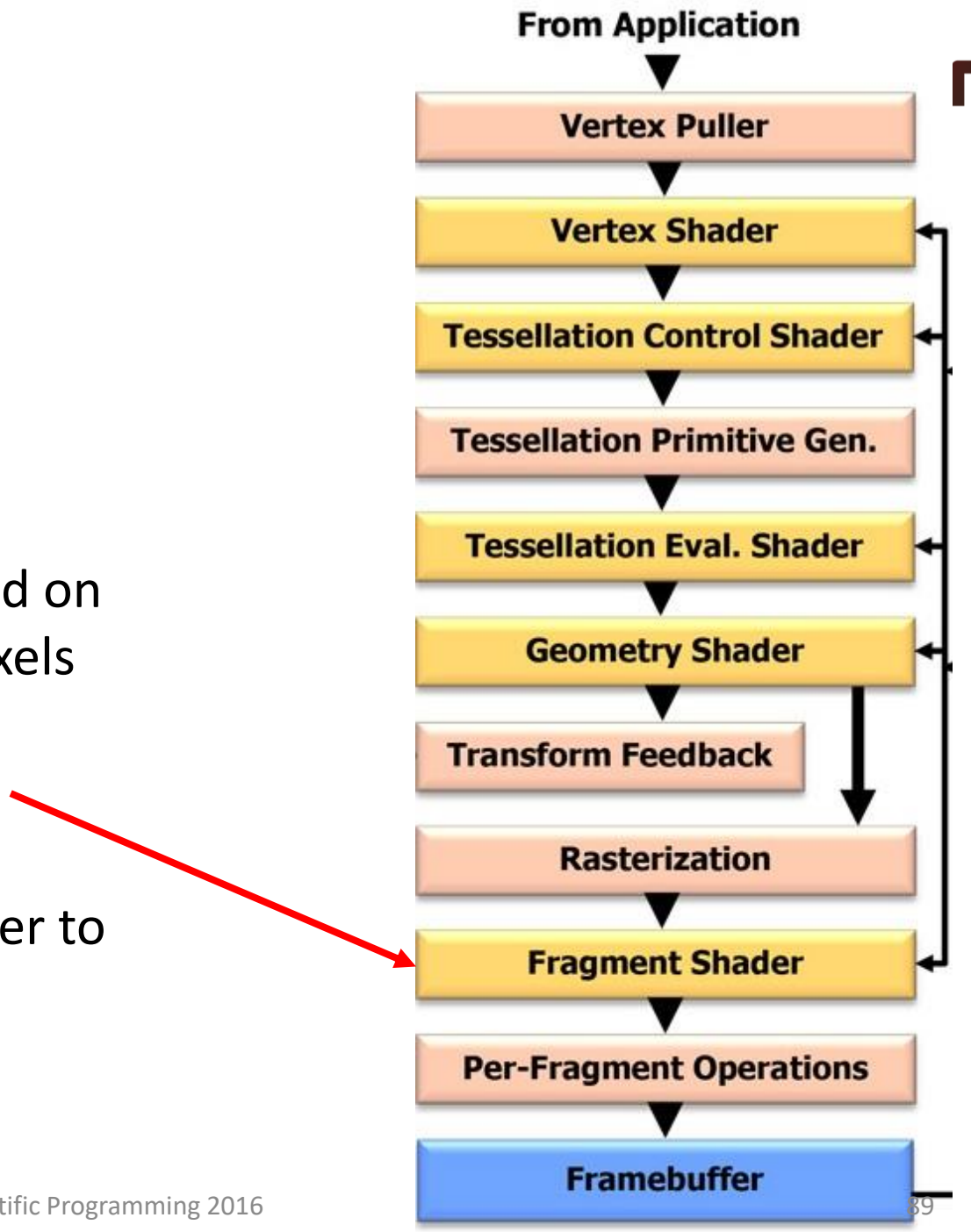


Programmable pipeline

Fragment (pixel) shader:

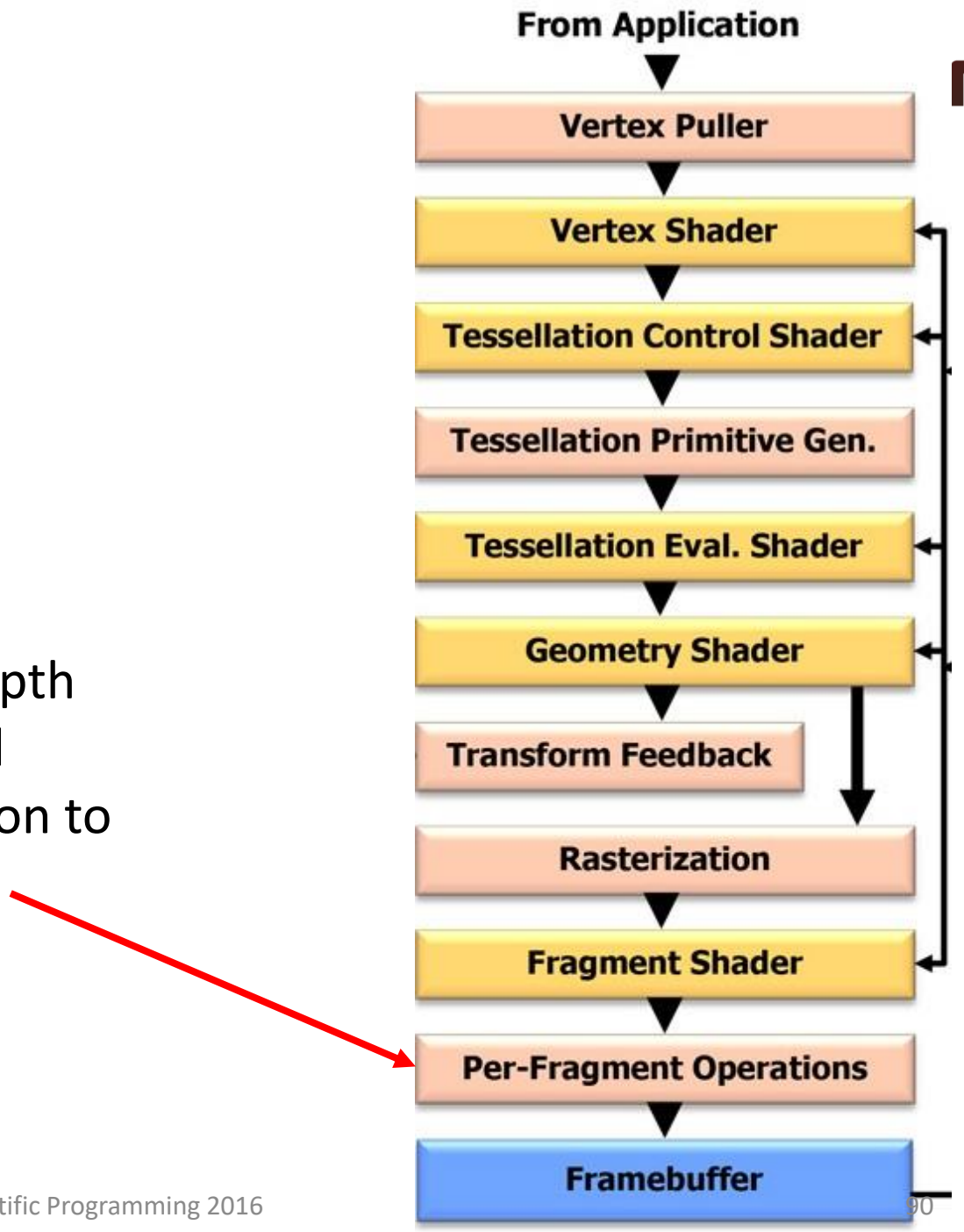
This programmable stage calculates based on an arbitrary program how to color the pixels of the rasterized primitives

This stage takes as arguments the interpolated vertex data, but can also refer to many 1-2-3D textures



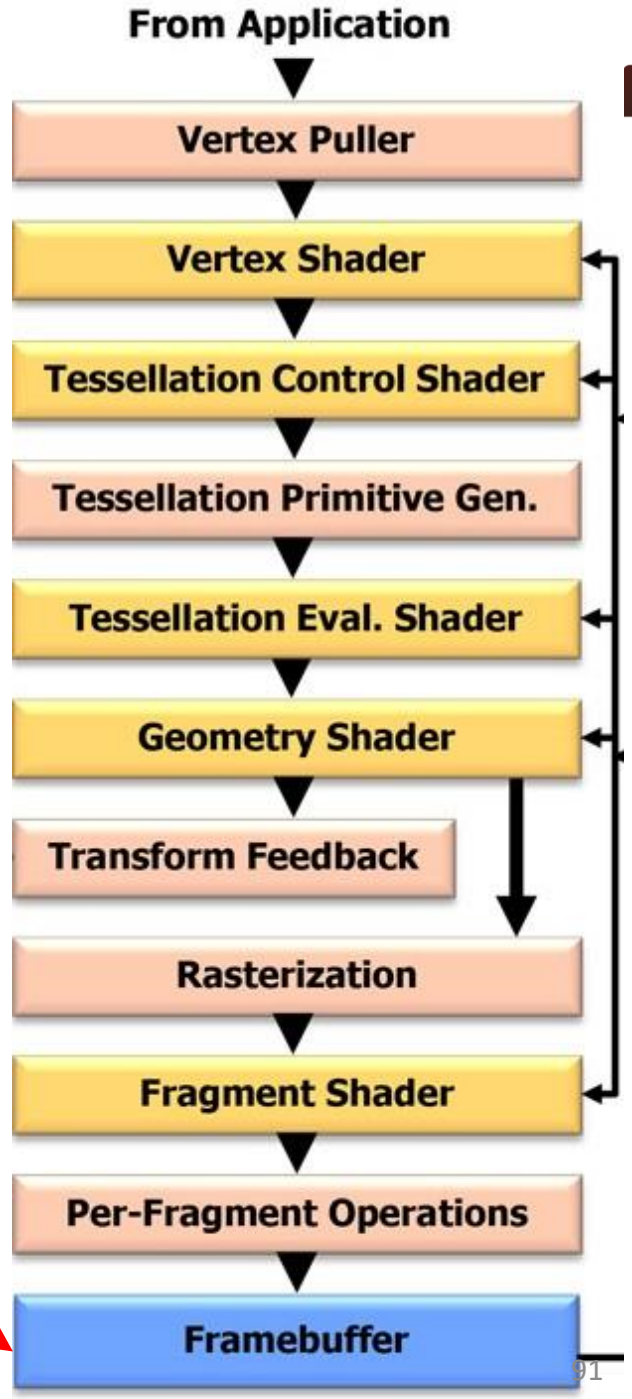
Programmable pipeline

Final per-fragment operations involve depth operations to calculate the occlusion and stencil operations to mask the composition to the target framebuffers



Programmable pipeline

One or more framebuffers are written
If they are linked to a screen, the image on the screen is updated in a next step



Programmable pipeline

Shader programs are simple text files, that are written in a dialect of C specialized for GPU operations

- It is a restricted version of C:
  - No library calls at all
  - No dynamical allocation
  - Just functions and structs

Example will be shown later

- It is an extended version of C:
  - Includes native support for 2-3-4 component vectors and matrices with all the common operations on them

Programming OpenGL

OpenGL interaction starts with obtaining an openGL context.

- This is a little bit complicated, we suggest to use a windowing library that hides this (and some other) complexity

- For example in SFML:

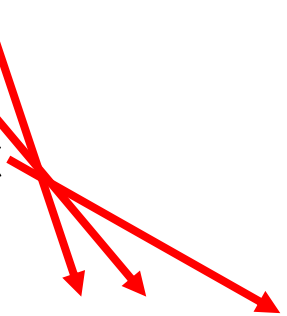
```
sf::ContextSettings settings(24, 8, 2, 3, 3);
sf::RenderWindow window(sf::VideoMode(800, 600),
                        "Title", sf::Style::Default, settings);
```

Programming OpenGL

OpenGL interaction starts with obtaining an openGL context.

- For example in SFML:

These are important openGL bits:
Depth 24 bits
Stencil 8 bits
Antialiasing 2x

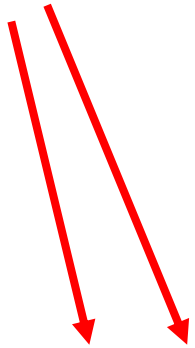


```
sf::ContextSettings settings(24, 8, 2, 4, 3);
sf::RenderWindow window(sf::VideoMode(800, 600),
                        "Title", sf::Style::Default, settings);
```

Programming OpenGL

OpenGL interaction starts with obtaining an openGL context.

OpenGL Version number: 4.3



- For example in SFML:

```
sf::ContextSettings settings(24, 8, 2, 4, 3);
sf::RenderWindow window(sf::VideoMode(800, 600),
                        "Title", sf::Style::Default, settings);
```

Programming OpenGL

OpenGL functions after version 1.0 are should be obtained by repeatedly querying them from a helper function:

They signature must be specified in the code matching the specification

```
typedef GLuint (*F_CreateShader) (GLenum shaderType);
```

On linux:

```
auto glCreateShader =
(F_CreateShader)glXGetProcAddress("glCreateShader");
```

On windows:

```
auto glCreateShader =
(F_CreateShader)wglGetProcAddress("glCreateShader");
```

Programming OpenGL

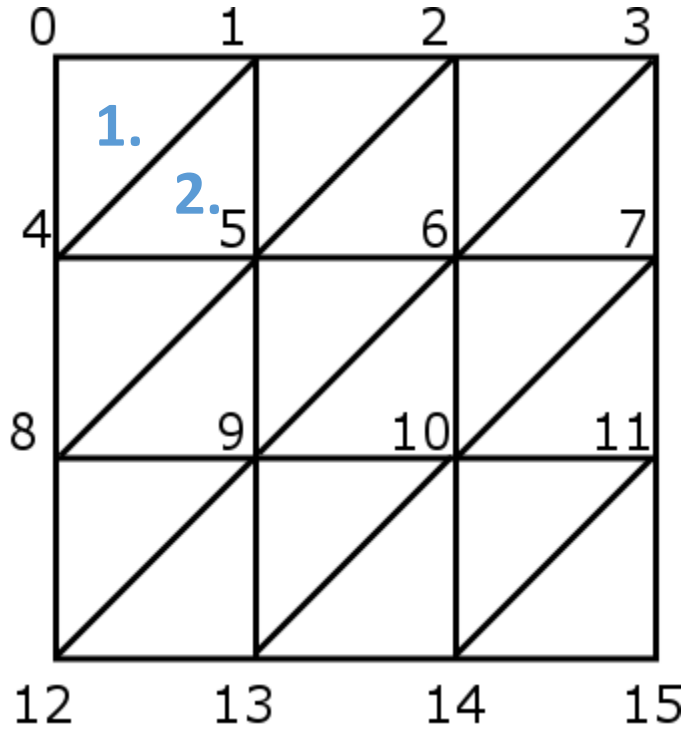
OpenGL functions after version 1.0 are should be obtained by repeatedly querying them from a helper function...

Or use a helper library for that: The OpenGL Extension Wrangler Library

Programming OpenGL

Describing a surface in a GPU friendly way:

- GPUs handle primitives: basically lines and triangles
  - Any surface should be made up with triangles
  - Now we can specify the triangles by writing the vertex coordinates in a list:
  - x0 y0, x1 y1, x4 y4 – **first triangle**
  - x4 y4, x1 y1, x5 y5 – **second triangle**
  - …



Programming OpenGL

Describing a surface in a GPU friendly way:

- Performance is limited by data transfer bottlenecks
  - The less data you need to move around, the faster the rendering will get!
- For this reason indexed rendering was invented!
  - We generate 2 lists:
    - the first containing the vertex data: only 1 entry for 1 vertex
    - An index list describing which vertices make up one primitive

Programming OpenGL

Describing a surface in a GPU friendly way:

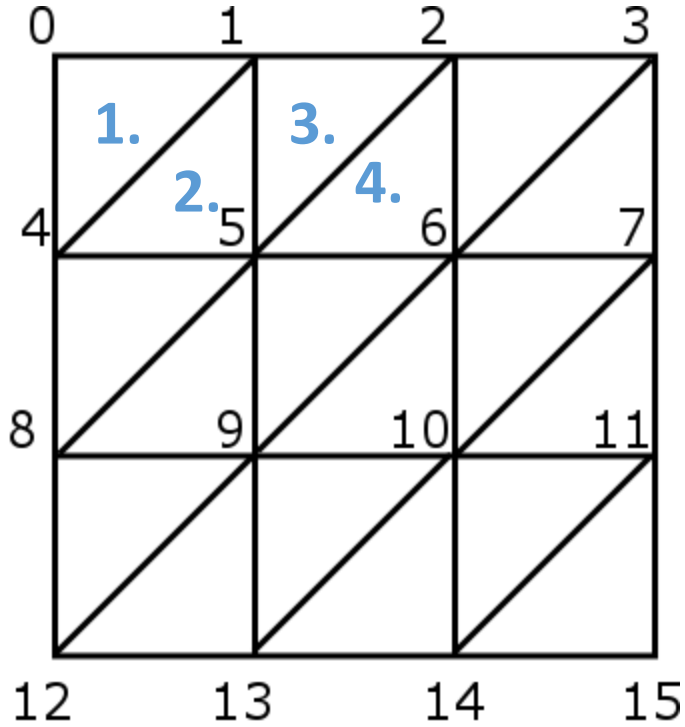
- Indexed description:

Buffer 1: Vertices

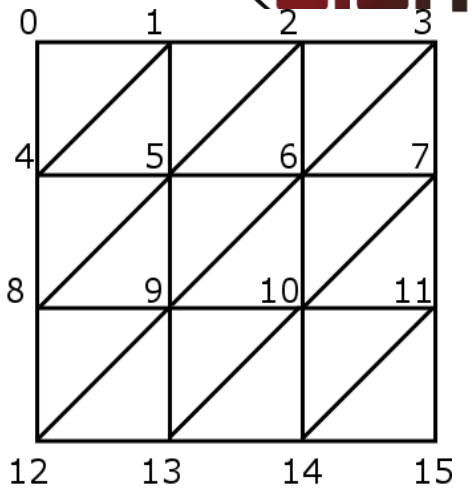
- x0 y0
- x1 y1
- x2 y2
- ….

Buffer 2: Indices

- 0 1 4 – 1. triangle
- 4 1 5 – 2. triangle
- 1 2 5 – 3. triangle
- 5 2 6 – 4. triangle



Programming OpenGL



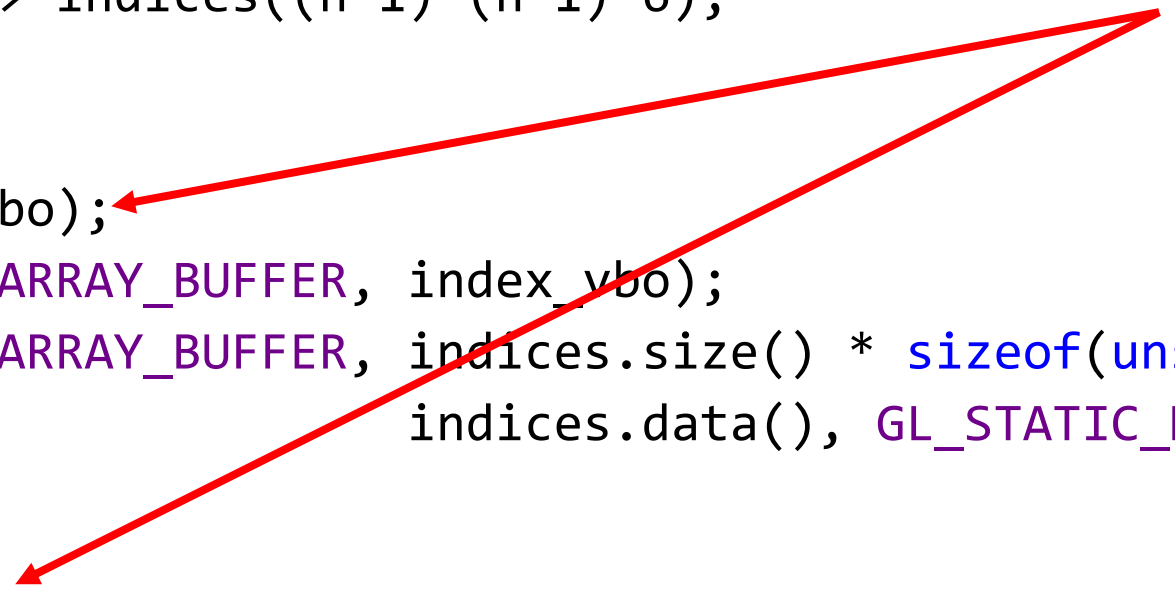
Describing an indexed triangle set in OpenGL:

```
std::vector<sf::Vector3f> vertices(n*n);
std::vector<unsigned int> indices((n-1)*(n-1)*6);

GLuint index_vbo;
glGenBuffers(1, &index_vbo);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, index_vbo);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices.size() * sizeof(unsigned int),
                                      indices.data(), GL_STATIC_DRAW);

GLuint vertices_vbo;
glGenBuffers(1, &vertices_vbo);
glBindBuffer(GL_ARRAY_BUFFER, vertices_vbo);
glBufferData(GL_ARRAY_BUFFER, vertices.size() * sizeof(sf::Vector3f),
                              points.data(), GL_STATIC_DRAW);
```

Programming OpenGL

Describing an indexed triangle set in OpenGL:

```
std::vector<sf::Vector3f> vertices(n*n);
std::vector<unsigned int> indices((n-1)*(n-1)*6);

GLuint index_vbo;
glGenBuffers(1, &index_vbo);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, index_vbo);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices.size() * sizeof(unsigned int),
                                      indices.data(), GL_STATIC_DRAW);

GLuint vertices_vbo;
glGenBuffers(1, &vertices_vbo);
glBindBuffer(GL_ARRAY_BUFFER, vertices_vbo);
glBufferData(GL_ARRAY_BUFFER, vertices.size() * sizeof(sf::Vector3f),
                              points.data(), GL_STATIC_DRAW);
```

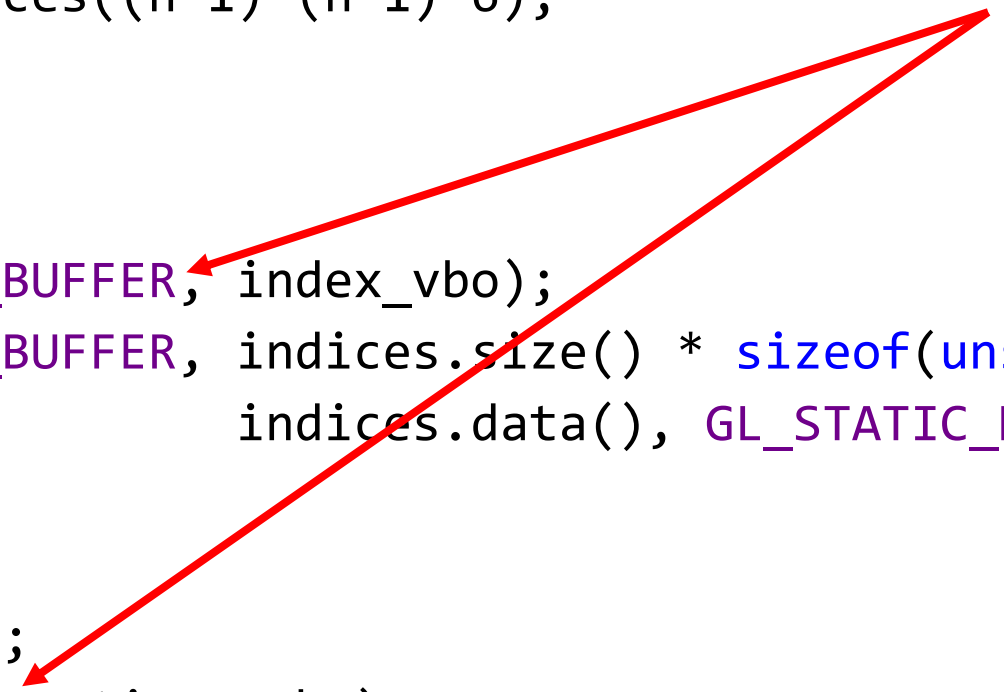


Programming OpenGL

Describing an indexed triangle set in OpenGL:

```
std::vector<sf::Vector3f> vertices(n*n);
std::vector<unsigned int> indices((n-1)*(n-1)*6);

GLuint index_vbo;
glGenBuffers(1, &index_vbo);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, index_vbo);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices.size() * sizeof(unsigned int),
                                      indices.data(), GL_STATIC_DRAW);

GLuint vertices_vbo;
glGenBuffers(1, &vertices_vbo);
glBindBuffer(GL_ARRAY_BUFFER, vertices_vbo);
glBufferData(GL_ARRAY_BUFFER, vertices.size() * sizeof(sf::Vector3f),
                              points.data(), GL_STATIC_DRAW);
```

Specify what kind of info we are going to store in them

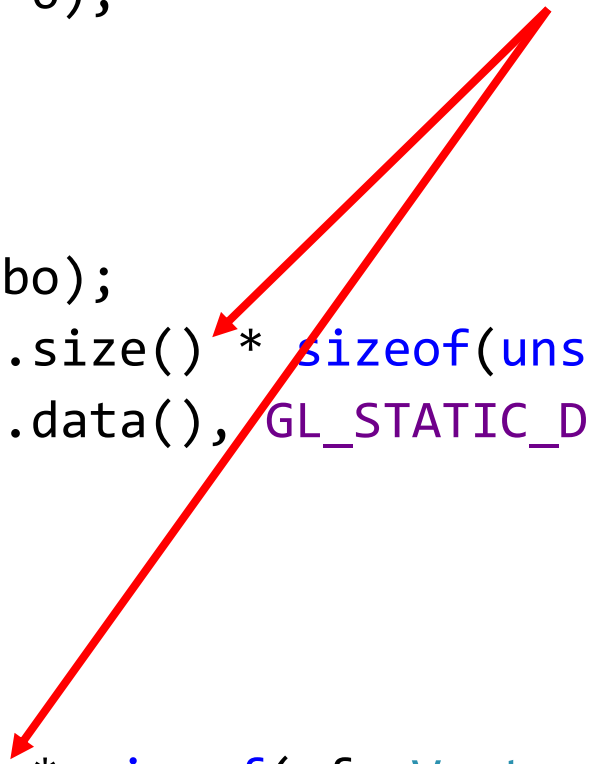


Programming OpenGL

Describing an indexed triangle set in OpenGL:

```
std::vector<sf::Vector3f> vertices(n*n);
std::vector<unsigned int> indices((n-1)*(n-1)*6);

GLuint index_vbo;
glGenBuffers(1, &index_vbo);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, index_vbo);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices.size() * sizeof(unsigned int),
                                      indices.data(), GL_STATIC_DRAW);

GLuint vertices_vbo;
glGenBuffers(1, &vertices_vbo);
glBindBuffer(GL_ARRAY_BUFFER, vertices_vbo);
glBufferData(GL_ARRAY_BUFFER, vertices.size() * sizeof(sf::Vector3f),
                              points.data(), GL_STATIC_DRAW);
```

Copy the data from the to `std::vectors` to OpenGL



Programming OpenGL

Describing the structure of our indexed triangle set in OpenGL:

```
GLuint vao;
glGenVertexArrays(1, &vao);
glBindVertexArray(vao);

glBindBuffer(GL_ARRAY_BUFFER, vertices_vbo);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, NULL);

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, index_vbo);

glEnableVertexAttribArray(0);
```

Programming OpenGL

Describing the structure of our indexed triangle set in OpenGL:

```
GLuint vao;
glGenVertexArrays(1, &vao);
glBindVertexArray(vao);

glBindBuffer(GL_ARRAY_BUFFER, vertices_vbo);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, NULL);

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, index_vbo);

glEnableVertexAttribArray(0);
```

Create a handle for the description of our data

Programming OpenGL

Describing the structure of our indexed triangle set in OpenGL:

```
GLuint vao;
glGenVertexArrays(1, &vao);
glBindVertexArray(vao);

glBindBuffer(GL_ARRAY_BUFFER, vertices_vbo);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, NULL);

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, index_vbo);

glEnableVertexAttribArray(0);
```

Link the vertex buffer we just created

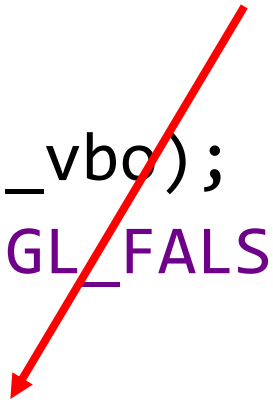
Specify the structure: 3 floats / vertex

Programming OpenGL

Describing the structure of our indexed triangle set in OpenGL:

```
GLuint vao;
glGenVertexArrays(1, &vao);
glBindVertexArray(vao);

glBindBuffer(GL_ARRAY_BUFFER, vertices_vbo);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, NULL);

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, index_vbo);

glEnableVertexAttribArray(0);
```

Link the index buffer too!



Programming OpenGL

Describing the structure of our indexed triangle set in OpenGL:

```
GLuint vao;
glGenVertexArrays(1, &vao);
glBindVertexArray(vao);

glBindBuffer(GL_ARRAY_BUFFER, vertices_vbo);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, NULL);

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, index_vbo);

glEnableVertexAttribArray(0);
```

Enable out vertex data input to the vertex shader

Programming OpenGL

Compiling our shaders:

```
std::ifstream vs_file("vs.glsl");
std::string vs_code{std::istreambuf_iterator<char>(vs_file),
                    std::istreambuf_iterator<char>()};

auto pvs_code = vs_code.c_str();
GLuint vs = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vs, 1, &pvs_code, NULL);
glCompileShader(vs);

GLuint shader = glCreateProgram();
glAttachShader(shader, fs);
glAttachShader(shader, vs);

glBindAttribLocation(shader, 0, "vertex_position");
glLinkProgram(shader);
```

Programming OpenGL

Compiling our shaders:

Load shader file to a string

```
std::ifstream vs_file("vs.glsl");
std::string vs_code{std::istreambuf_iterator<char>(vs_file),
                    std::istreambuf_iterator<char>()};

auto pvs_code = vs_code.c_str();
GLuint vs = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vs, 1, &pvs_code, NULL);
glCompileShader(vs);

GLuint shader = glCreateProgram();
glAttachShader(shader, fs);
glAttachShader(shader, vs);

glBindAttribLocation(shader, 0, "vertex_position");
glLinkProgram(shader);
```

Programming OpenGL

Compiling our shaders:

```
std::ifstream vs_file("vs.glsl");
std::string vs_code{std::istreambuf_iterator<char>(vs_file),
                    std::istreambuf_iterator<char>()};

auto pvs_code = vs_code.c_str();
GLuint vs = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vs, 1, &pvs_code, NULL);
glCompileShader(vs);

GLuint shader = glCreateProgram();
glAttachShader(shader, fs);
glAttachShader(shader, vs);

glBindAttribLocation(shader, 0, "vertex_position");
glLinkProgram(shader);
```

Send shader data string to OpenGL and compile it

(repeat these steps for a fragment shader too, omitted for shortness)

Programming OpenGL

Compiling our shaders:

```
std::ifstream vs_file("vs.glsl");
std::string vs_code{std::istreambuf_iterator<char>(vs_file),
                    std::istreambuf_iterator<char>()};

auto pvs_code = vs_code.c_str();
GLuint vs = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vs, 1, &pvs_code, NULL);
glCompileShader(vs);

GLuint shader = glCreateProgram();
glAttachShader(shader, fs);
glAttachShader(shader, vs);

glBindAttribLocation(shader, 0, "vertex_position");
glLinkProgram(shader);
```

Create “the” shader object that groups together the vertex and fragment shaders

Programming OpenGL

Compiling our shaders:

```
std::ifstream vs_file("vs.glsl");
std::string vs_code{std::istreambuf_iterator<char>(vs_file),
                    std::istreambuf_iterator<char>()};

auto pvs_code = vs_code.c_str();
GLuint vs = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vs, 1, &pvs_code, NULL);
glCompileShader(vs);

GLuint shader = glCreateProgram();
glAttachShader(shader, fs);
glAttachShader(shader, vs);

glBindAttribLocation(shader, 0, "vertex_position");
glLinkProgram(shader);
```

Specify an input slot and name for vertex data

Finish setup

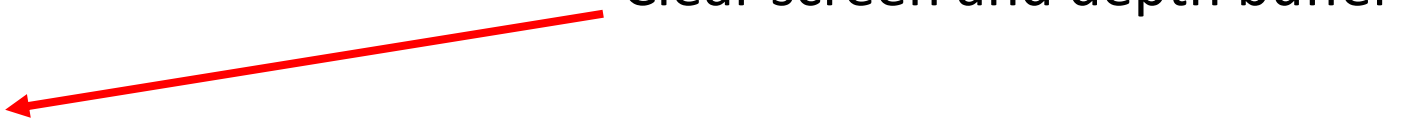
Programming OpenGL

Rendering our triangles:

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glUseProgram(shader);
glBindVertexArray(vao);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, index_vbo);
glDrawElements(GL_TRIANGLES, (n-1)*(n-1)*6,
                        GL_UNSIGNED_INT, (void*)0);
```

Programming OpenGL

Rendering our triangles:




```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glUseProgram(shader);
glBindVertexArray(vao);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, index_vbo);
glDrawElements(GL_TRIANGLES, (n-1)*(n-1)*6,
                        GL_UNSIGNED_INT, (void*)0);
```

Programming OpenGL

Rendering our triangles:

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glUseProgram(shader);
glBindVertexArray(vao);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, index_vbo);
glDrawElements(GL_TRIANGLES, (n-1)*(n-1)*6,
                        GL_UNSIGNED_INT, (void*)0);
```

Set active shader

Programming OpenGL

Rendering our triangles:

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glUseProgram(shader);
glBindVertexArray(vao);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, index_vbo);
glDrawElements(GL_TRIANGLES, (n-1)*(n-1)*6,
                        GL_UNSIGNED_INT, (void*)0);
```

Set active vertex description and buffer

Programming OpenGL

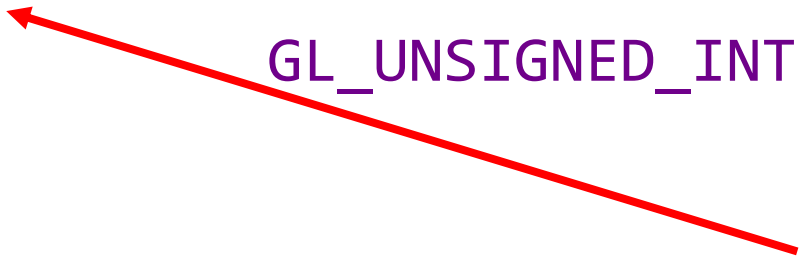
Rendering our triangles:

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glUseProgram(shader);
glBindVertexArray(vao);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, index_vbo);
glDrawElements(GL_TRIANGLES, (n-1)*(n-1)*6,
                        GL_UNSIGNED_INT, (void*)0);
```

Programming OpenGL

Rendering our triangles:

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glUseProgram(shader);
glBindVertexArray(vao);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, index_vbo);
glDrawElements(GL_TRIANGLES, (n-1)*(n-1)*6,
                              GL_UNSIGNED_INT, (void*)0);
```



Draw the triangles! ☺

Programming OpenGL

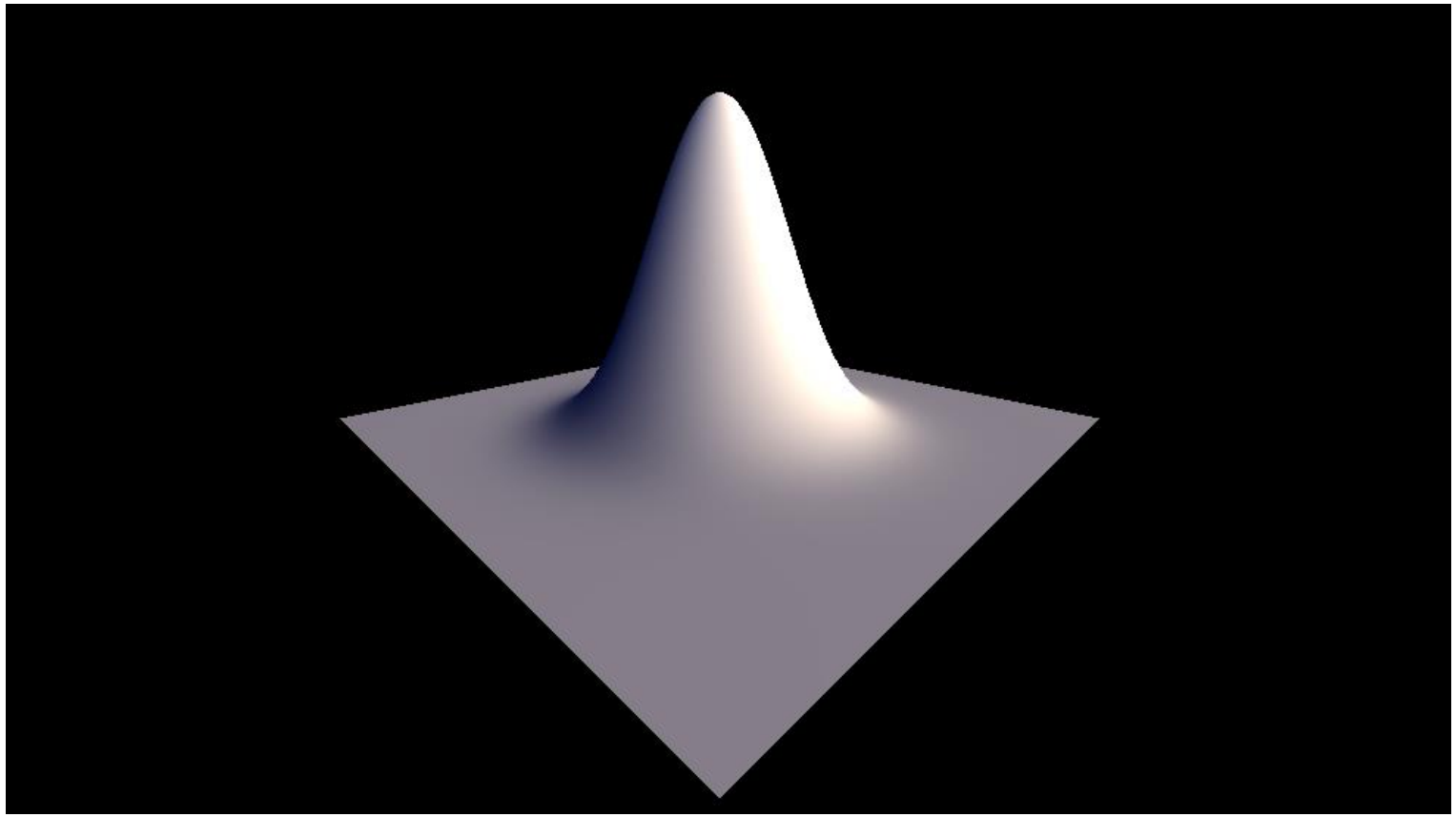
Short summary of steps:

- Buffer setup
  - Generating triangles
  - Specifying vertex layout
- Shader setup
  - Load file, create, compile, link shaders
- Render step
  - Set active shader, buffer, issue a draw call

This is done repeatedly in the window's event loop!

Minimal lighting example

Now comes the mathematics, because we'd like to draw this:



Projective space – Affine Transformations

3D matrices can represent some, but not all of the wanted transformations:

- Rotations
- Scalings
- Skews

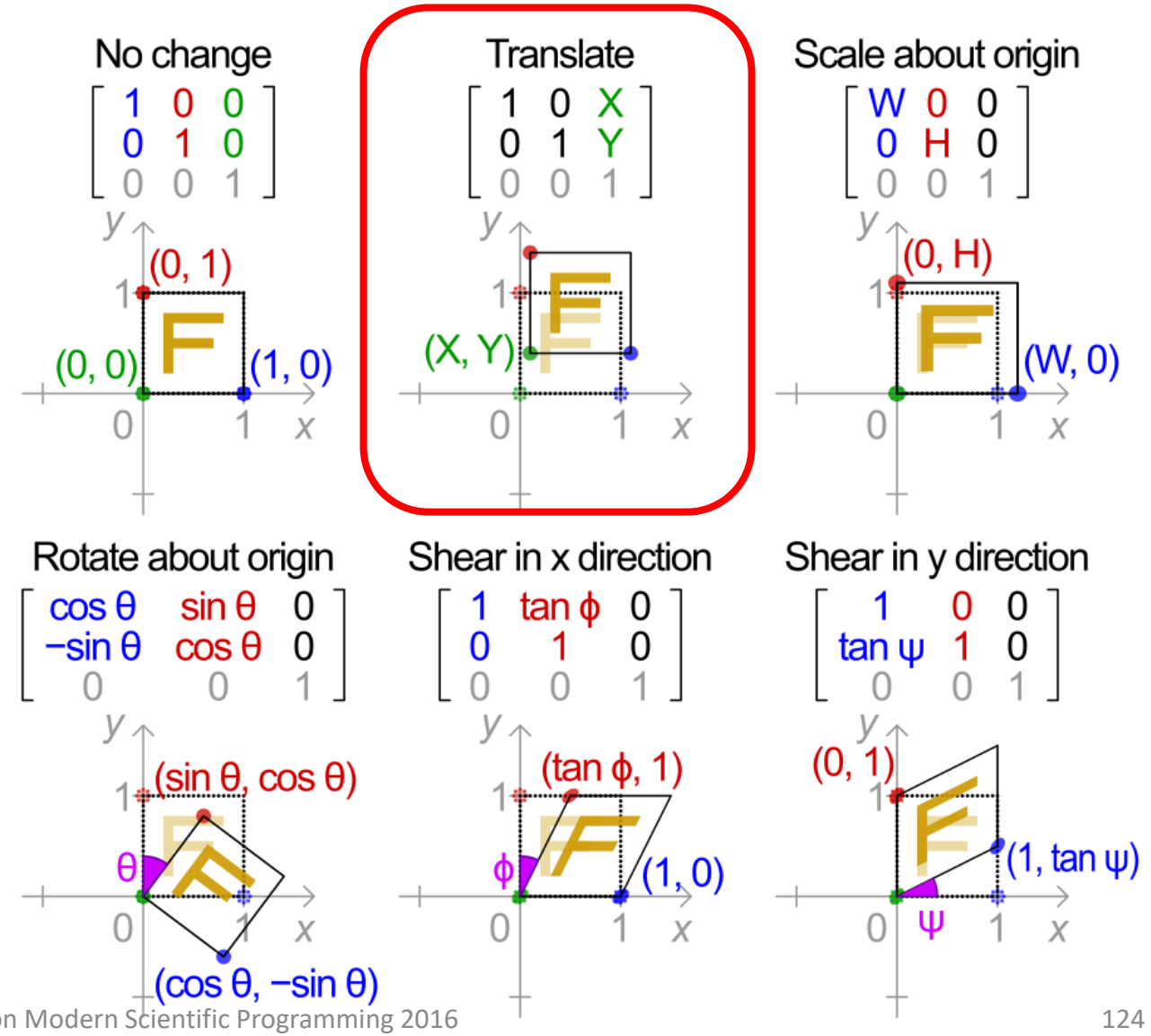
But what about two extremely important ones:

- Translations
- Projections

Projective space – Affine Transformations

Switching to projective space we can represent affine transformations too.

For example in 2D:



Projective space – Affine Transformations

We can describe a perspective projection to a 2D plane by:

$$\begin{bmatrix} X \\ Y \\ Z \\ W \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & D^{-1} & 0 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Projective space – Affine Transformations

We can describe a perspective projection to a 2D plane by:

Input 3D coordinates

$$\begin{bmatrix} X \\ Y \\ Z \\ W \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & D^{-1} & 0 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Projective space – Affine Transformations

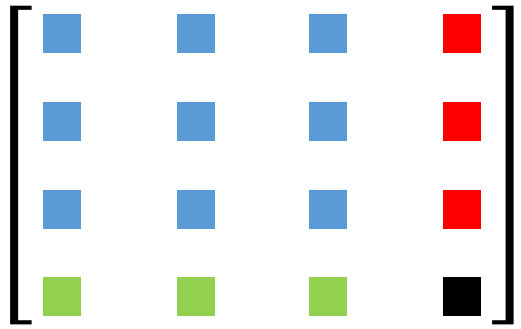
We can describe a perspective projection to a 2D plane by:

$$\begin{bmatrix} X \\ Y \\ Z \\ W \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & D^{-1} & 0 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Projected coordinates, of course, usually only X, Y are relevant usually

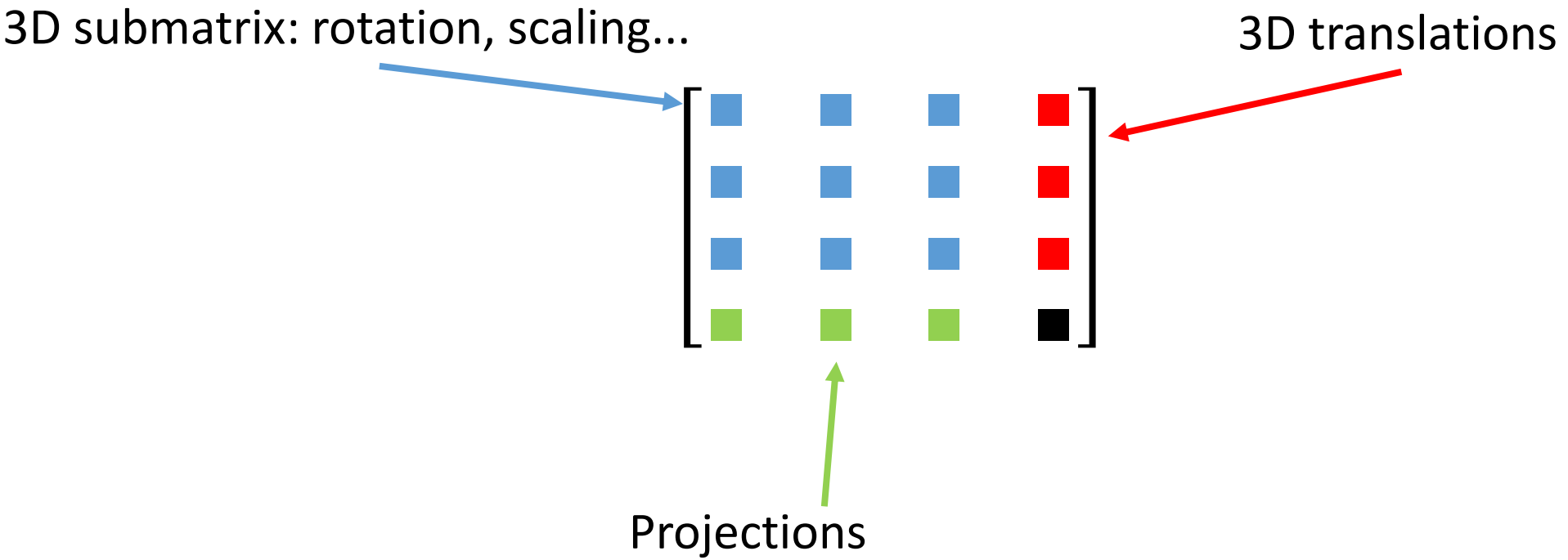
Projective space – Affine Transformations

So all wanted transformations can be represented by a 4x4 matrix:



Projective space – Affine Transformations

So all wanted transformations can be represented by a 4x4 matrix:



World-View-Projection Transformations

The typical series of transformations applied to an object during 3D rendering are the following:

- World transformation:
  - Transforming the object from its local coordinates to its real placement in our 3D scene (translations, rotation, scaling, etc...)
- View transformation:
  - Transforming to the camera coordinate system
- Projection:
  - Project to the screen space

The View transform and the camera

The camera describes the viewing point and angles in the 3D scene, conveniently with 3 vectors:

- Camera position vector (p)
- The target point to be looked at (t)
- The up-vector (u)

The local coordinate axes for the camera with these:

$$\vec{z} = \frac{\vec{p}-\vec{t}}{|\vec{p}-\vec{t}|} \qquad \vec{x} = \frac{\vec{u}\times\vec{z}}{|\vec{u}\times\vec{z}|} \qquad \vec{y} = \vec{z}\times\vec{x}$$

The View transform and the camera

And the view matrix with the previous axes and the position vector:

$$\begin{bmatrix} \vec{x}.x & \vec{y}.x & \vec{z}.x & 0 \\ \vec{x}.y & \vec{y}.y & \vec{z}.y & 0 \\ \vec{x}.z & \vec{y}.z & \vec{z}.z & 0 \\ -\vec{x}\cdot\vec{p} & -\vec{y}\cdot\vec{p} & -\vec{z}\cdot\vec{p} & 1 \end{bmatrix}$$

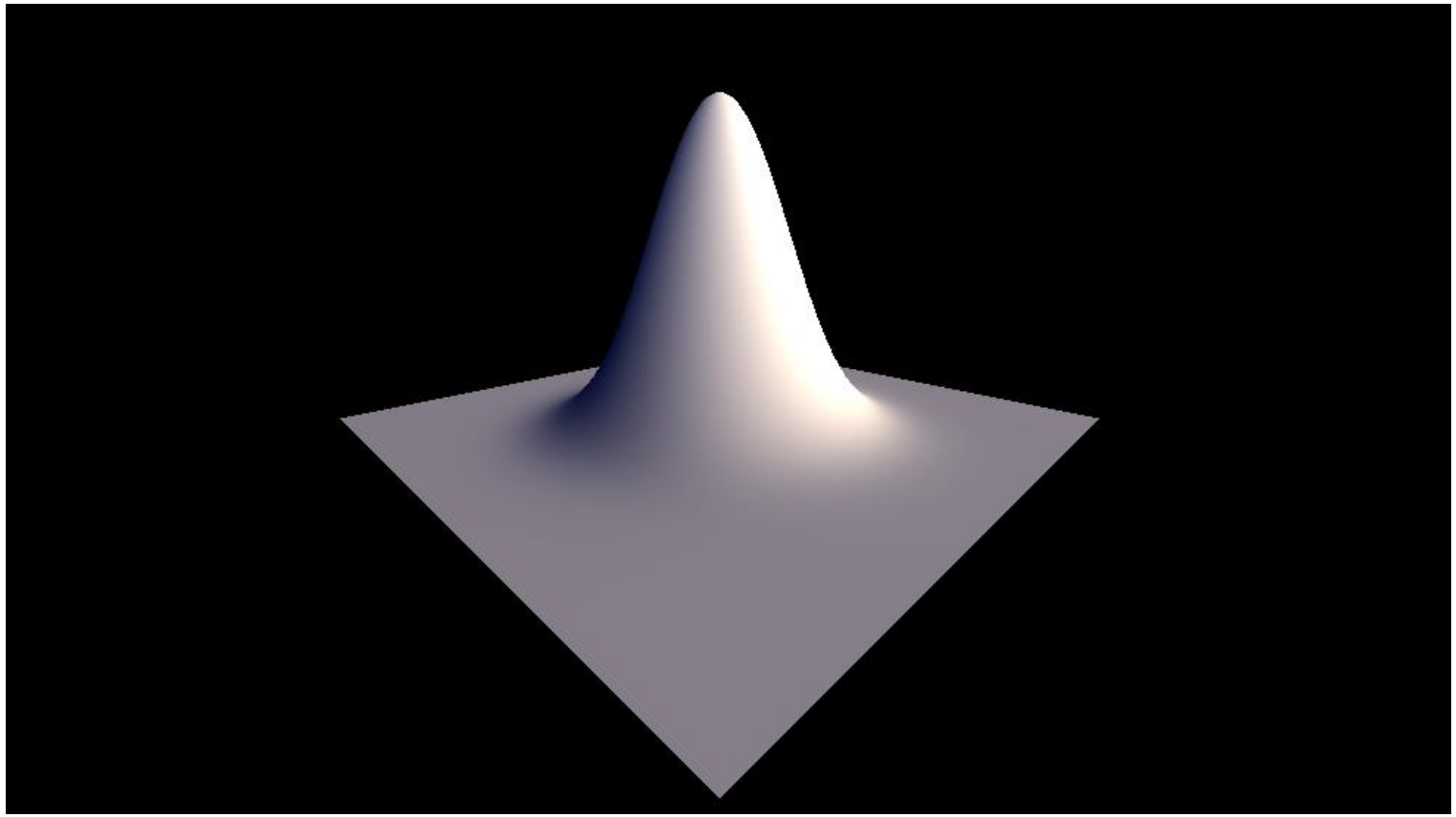
Typical geometry processing

In a typical scenario:

- The transformation matrices are calculated, multiplied, updated on the host side
- Each frame (or on change) the final MVP matrix is uploaded to the GPU
- In the vertex shader positions and normals are transformed by MVP matrix
- Sometimes for more precise lighting the normal transformation is done in the fragment(pixel) shader

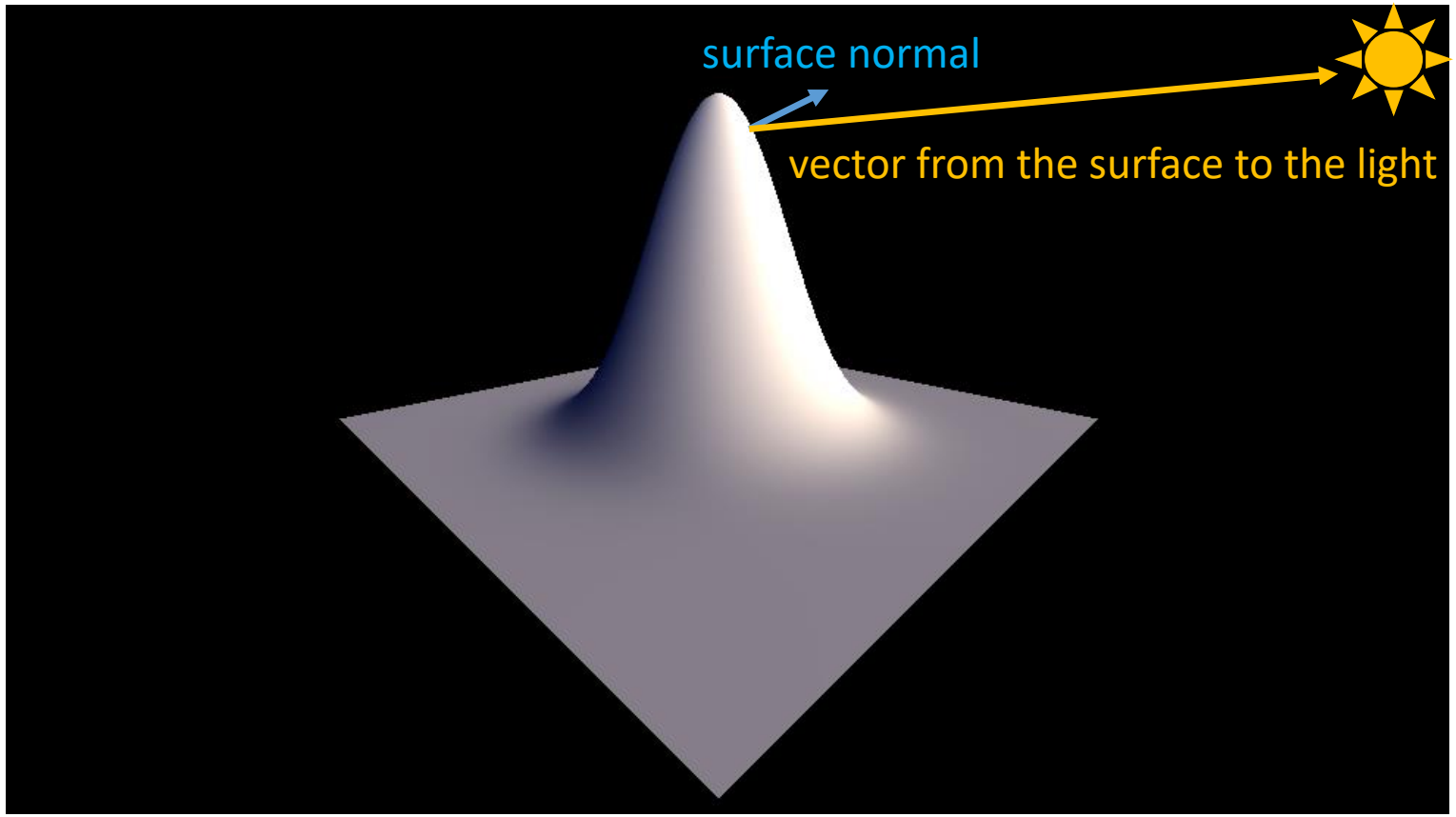
Minimal lighting example

As a simple illustration lets review a point lighting scheme:



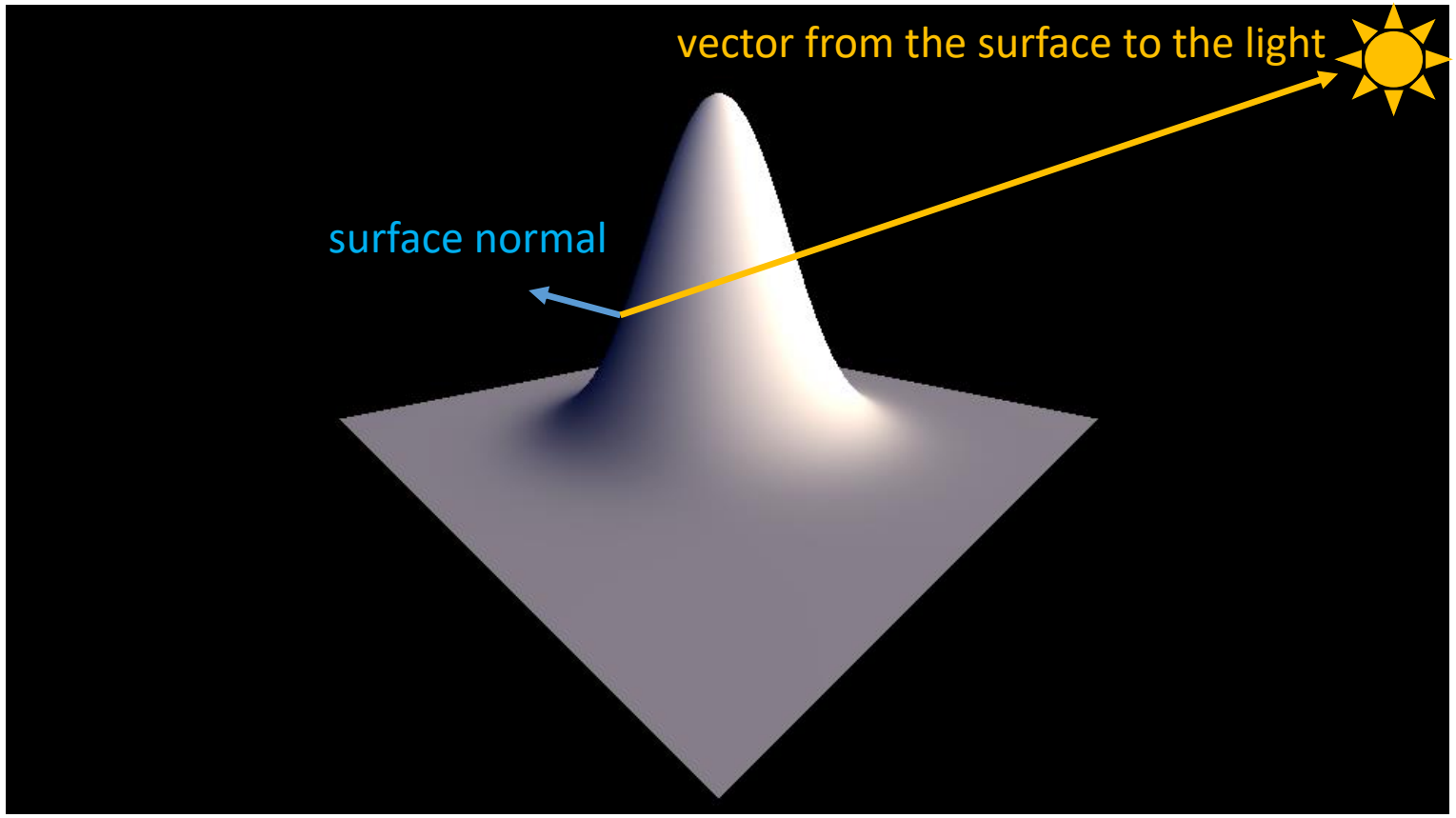
Minimal lighting example

If the normal is pointing towards the light, make the surface bright



Minimal lighting example

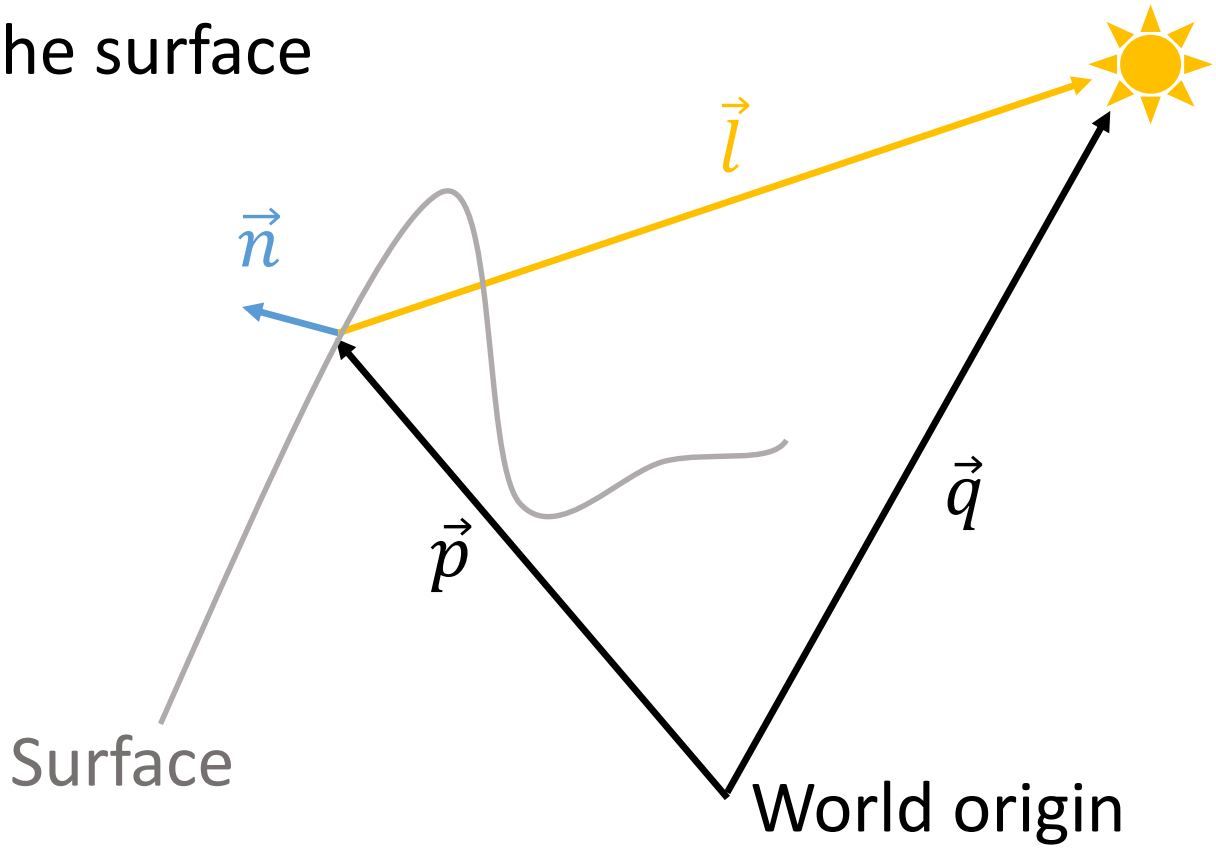
If the normal is pointing away, make surface dark



Minimal lighting example

Introduce the notation:

- $\vec{p}$ is the position of a point on the surface
- $\vec{n}$ is the surface normal at $\vec{p}$
- $\vec{q}$ is the light position
- $\vec{l} = \vec{q} - \vec{p}$ is the vector from the surface to the light



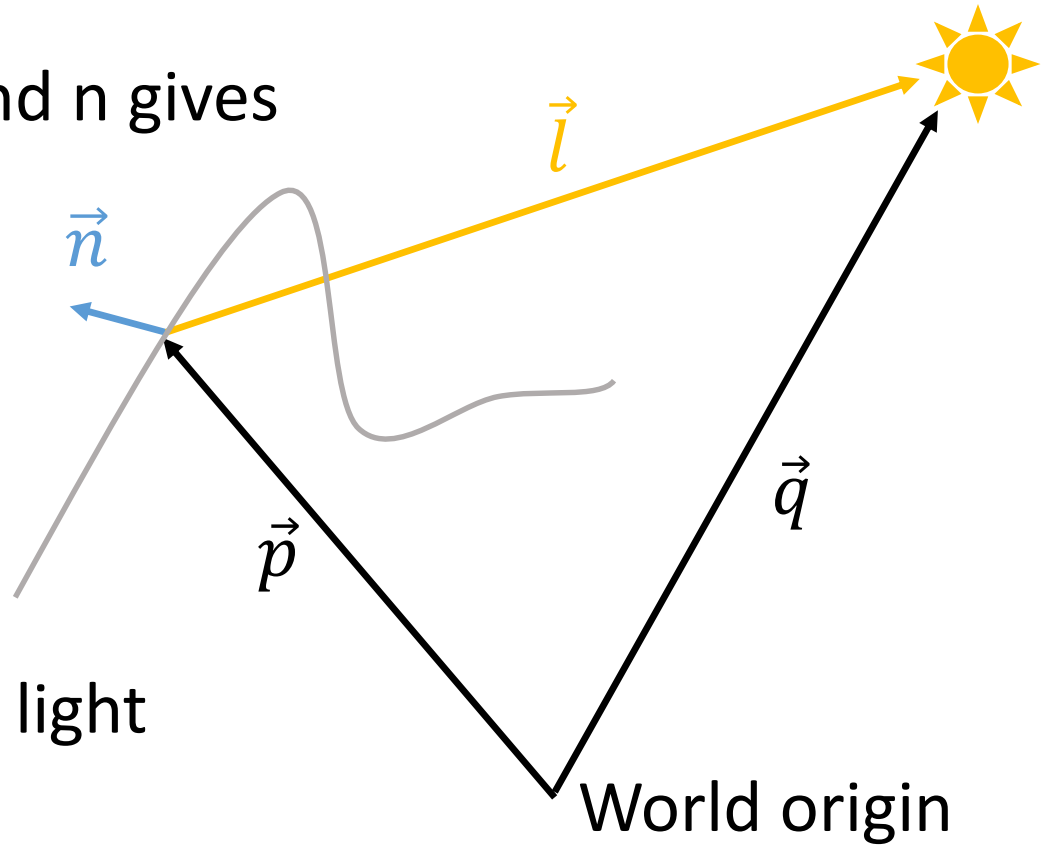
Minimal lighting example

Now simply, the dot product between l and n gives the desired shading scalar:

$$d = \frac{\vec{l}}{|\vec{l}|} \cdot \vec{n}$$

$-1 \leq d < 0$ we're looking away from the light

$0 < d \leq 1$ we're looking toward the light



Minimal lighting example

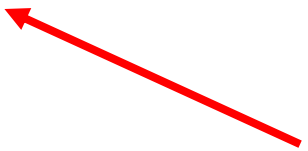
The vertex shader looks like this:

```
layout(location = 0) in vec3 vertex_position;
layout(location = 1) in vec3 vertex_normal;

uniform mat4 w, vp;
out vec4 pos, normal_pos;

void main(){
    pos = vec4(vertex_position, 1.0) * w;
    gl_Position = pos * vp;
    normal_pos = vec4(vertex_position + vertex_normal, 1.0) * w;
}
```

Minimal lighting example

The vertex shader looks like this:

```
layout(location = 0) in vec3 vertex_position;
layout(location = 1) in vec3 vertex_normal;

uniform mat4 w, vp;
out vec4 pos, normal_pos;

void main(){
    pos = vec4(vertex_position, 1.0) * w;
    gl_Position = pos * vp;
    normal_pos = vec4(vertex_position + vertex_normal, 1.0) * w;
}
```



Definition of the vertex layout:
3 components for positions
3 components for normals

Minimal lighting example

The vertex shader looks like this:

```
layout(location = 0) in vec3 vertex_position;
layout(location = 1) in vec3 vertex_normal;

uniform mat4 w, vp;
out vec4 pos, normal_pos;

void main(){
    pos = vec4(vertex_position, 1.0) * w;
    gl_Position = pos * vp;
    normal_pos = vec4(vertex_position + vertex_normal, 1.0) * w;
}
```

Global shader variables set from host:
The world and the view-projection matrices

Minimal lighting example

The vertex shader looks like this:

```
layout(location = 0) in vec3 vertex_position;
layout(location = 1) in vec3 vertex_normal;

uniform mat4 w, vp;
out vec4 pos, normal_pos;

void main(){
    pos = vec4(vertex_position, 1.0) * w;
    gl_Position = pos * vp;
    normal_pos = vec4(vertex_position + vertex_normal, 1.0) * w;
}
```

Minimal lighting example

The vertex shader looks like this:

```
layout(location = 0) in vec3 vertex_position;
layout(location = 1) in vec3 vertex_normal;

uniform mat4 w, vp;
out vec4 pos, normal_pos;

void main(){
    pos = vec4(vertex_position, 1.0) * w;
    gl_Position = pos * vp;
    normal_pos = vec4(vertex_position + vertex_normal, 1.0) * w;
}
```

Minimal lighting example

The vertex shader looks like this:

```
layout(location = 0) in vec3 vertex_position;
layout(location = 1) in vec3 vertex_normal;

uniform mat4 w, vp;
out vec4 pos, normal_pos;

void main(){
    pos = vec4(vertex_position, 1.0) * w;
    gl_Position = pos * vp;
    normal_pos = vec4(vertex_position + vertex_normal, 1.0) * w;
}
```

First transform the vertex with the world matrix

Minimal lighting example

The vertex shader looks like this:

```
layout(location = 0) in vec3 vertex_position;
layout(location = 1) in vec3 vertex_normal;

uniform mat4 w, vp;
out vec4 pos, normal_pos;

void main(){
    pos = vec4(vertex_position, 1.0) * w;
    gl_Position = pos * vp;
    normal_pos = vec4(vertex_position + vertex_normal, 1.0) * w;
}
```

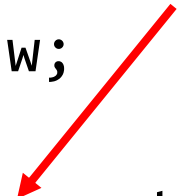
Then transform with the view-projection matrix
and send it to the special variable:
the vertex position output

Minimal lighting example

The vertex shader looks like this:

```
layout(location = 0) in vec3 vertex_position;
layout(location = 1) in vec3 vertex_normal;

uniform mat4 w, vp;
out vec4 pos, normal_pos;

void main(){
    pos = vec4(vertex_position, 1.0) * w;
    gl_Position = pos * vp;
    normal_pos = vec4(vertex_position + vertex_normal, 1.0) * w;
}
```

Transform the normal to a position vector and write to the output variable



Minimal lighting example

The fragment (pixel) shader looks like this:

```
in vec4 pos, normal_pos;
out vec4 color;

void main(){
    vec4 light_pos = vec4(5.0, 5.0, 5.0, 1.0);
    float d = dot( normal_pos - pos,
                   normalize(light_pos - pos));
    vec3 col = vec3(0.1, 0.1, 0.2) +
               0.8 * d * vec3(1.0f, 0.9f, 0.7f);
    color = vec4(col, 1.0);
}
```

Minimal lighting example

The fragment (pixel) shader looks like this:

Input data from the vertex shader

```
in vec4 pos, normal_pos;
out vec4 color;

void main(){
    vec4 light_pos = vec4(5.0, 5.0, 5.0, 1.0);
    float d = dot( normal_pos - pos,
                   normalize(light_pos - pos));
    vec3 col = vec3(0.1, 0.1, 0.2) +
               0.8 * d * vec3(1.0f, 0.9f, 0.7f);
    color = vec4(col, 1.0);
}
```

Minimal lighting example

The fragment (pixel) shader looks like this:

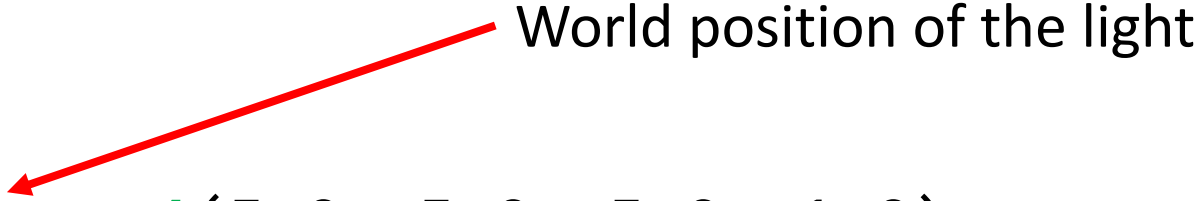
```
in vec4 pos, normal_pos;
out vec4 color;

void main(){
    vec4 light_pos = vec4(5.0, 5.0, 5.0, 1.0);
    float d = dot( normal_pos - pos,
                   normalize(light_pos - pos));
    vec3 col = vec3(0.1, 0.1, 0.2) +
               0.8 * d * vec3(1.0f, 0.9f, 0.7f);
    color = vec4(col, 1.0);
}
```

Color output of the fragment shader

Minimal lighting example

The fragment (pixel) shader looks like this:

```
in vec4 pos, normal_pos;
out vec4 color;

void main(){
    vec4 light_pos = vec4(5.0, 5.0, 5.0, 1.0);
    float d = dot( normal_pos - pos,
                   normalize(light_pos - pos));
    vec3 col = vec3(0.1, 0.1, 0.2) +
               0.8 * d * vec3(1.0f, 0.9f, 0.7f);
    color = vec4(col, 1.0);
}
```



Minimal lighting example

The fragment (pixel) shader looks like this:

```
in vec4 pos, normal_pos;
out vec4 color;

void main(){
    vec4 light_pos = vec4(5.0, 5.0, 5.0, 1.0);
    float d = dot( normal_pos - pos,
                   normalize(light_pos - pos));
    vec3 col = vec3(0.1, 0.1, 0.2) +
               0.8 * d * vec3(1.0f, 0.9f, 0.7f);
    color = vec4(col, 1.0);
}
```

Create the normalized direction vectors and calculate their dot product

Minimal lighting example

The fragment (pixel) shader looks like this:

```
in vec4 pos, normal_pos;
out vec4 color;

void main(){
    vec4 light_pos = vec4(5.0, 5.0, 5.0, 1.0);
    float d = dot( normal_pos - pos,
                   normalize(light_pos - pos));
    vec3 col = vec3(0.1, 0.1, 0.2) +
               0.8 * d * vec3(1.0f, 0.9f, 0.7f);
    color = vec4(col, 1.0);
}
```

Some nice color blending

Minimal lighting example

The fragment (pixel) shader looks like this:

```
in vec4 pos, normal_pos;
out vec4 color;

void main(){
    vec4 light_pos = vec4(5.0, 5.0, 5.0, 1.0);
    float d = dot( normal_pos - pos,
                   normalize(light_pos - pos));
    vec3 col = vec3(0.1, 0.1, 0.2) +
               0.8 * d * vec3(1.0f, 0.9f, 0.7f);
    color = vec4(col, 1.0);
}
```

Set final output color

Minimal lighting example

The final code is available online!