h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
fbi
FACHBEREICH INFORMATIK

Visual Computing

Graphic objects and their programming

Darmstadt University

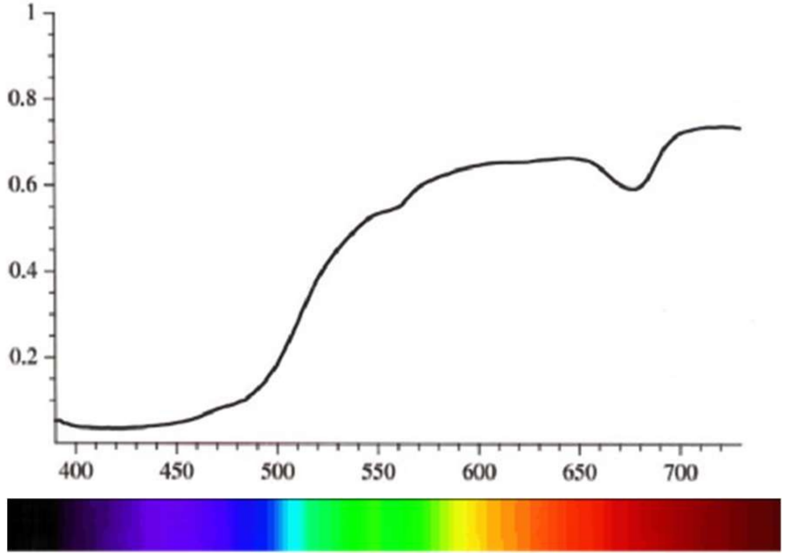
Prof. Dr Elke Hergenröther
Björn Frömmer
Prof. Dr Benjamin Meyer

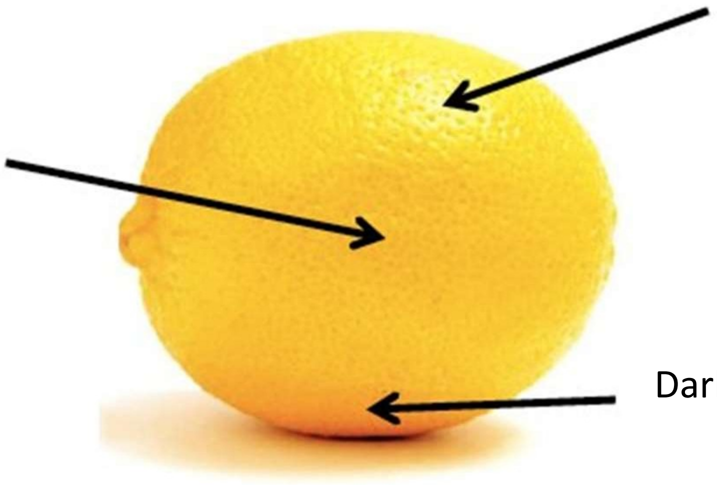
CHAPTER 3

Colours and primitives

Colour spaces

- What is colour and how would you store it?
- Physics: Reflected light



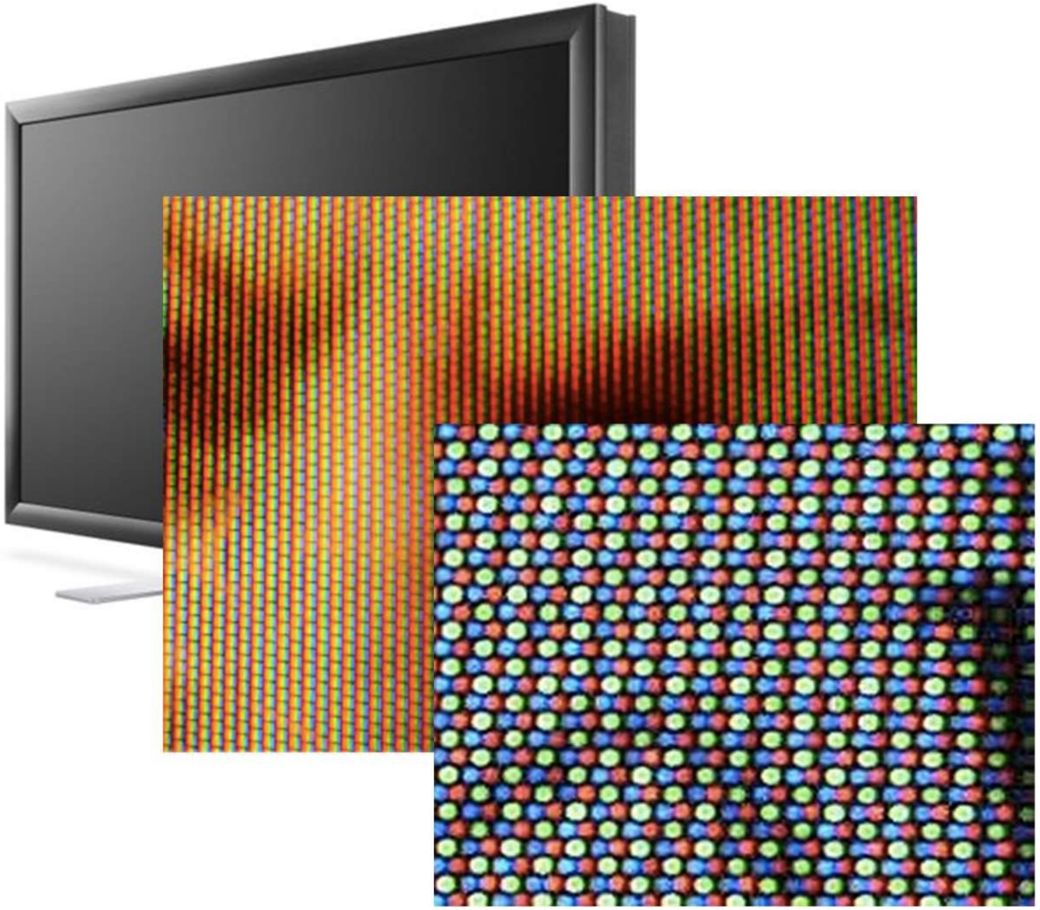


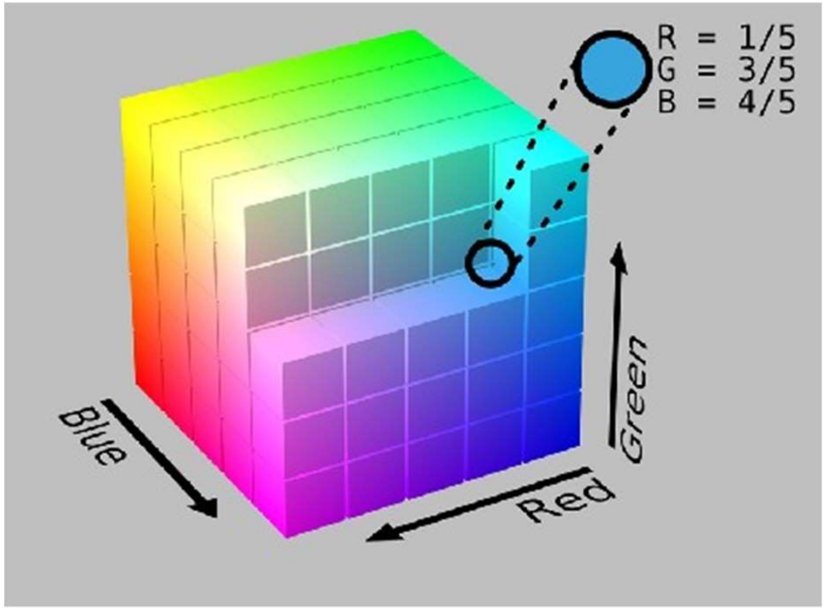
Blue light absorbed

Speculative highlights

Darkened areas

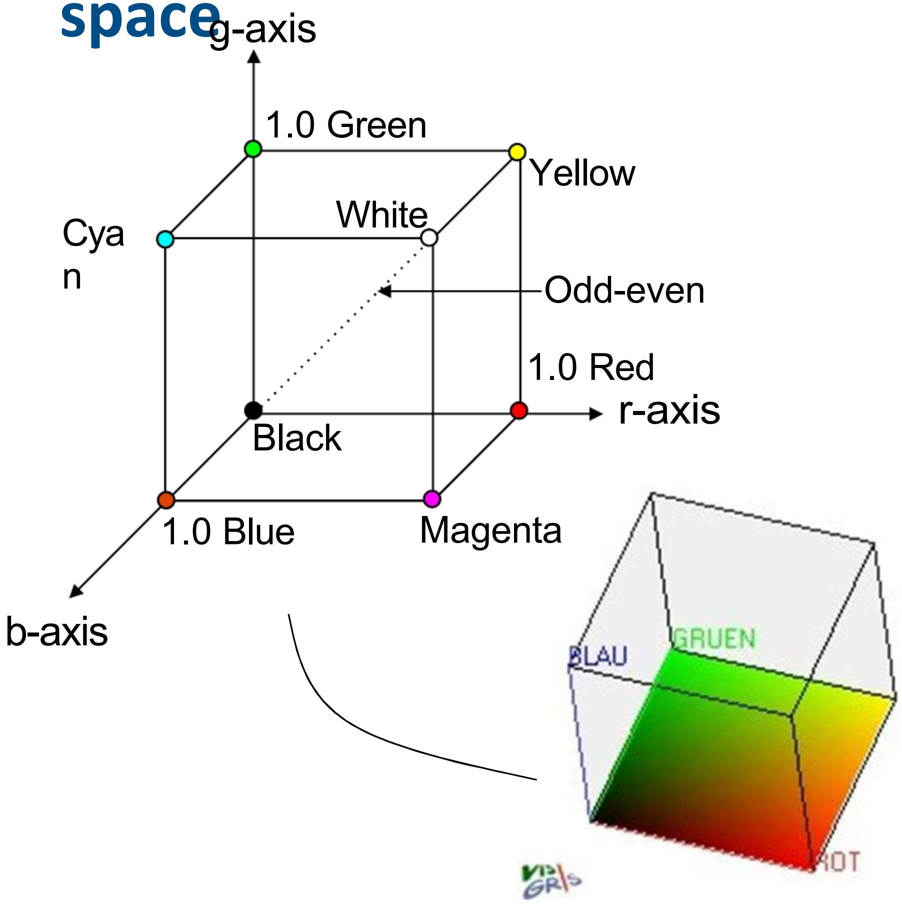
Colour spaces





RGB Colour Space

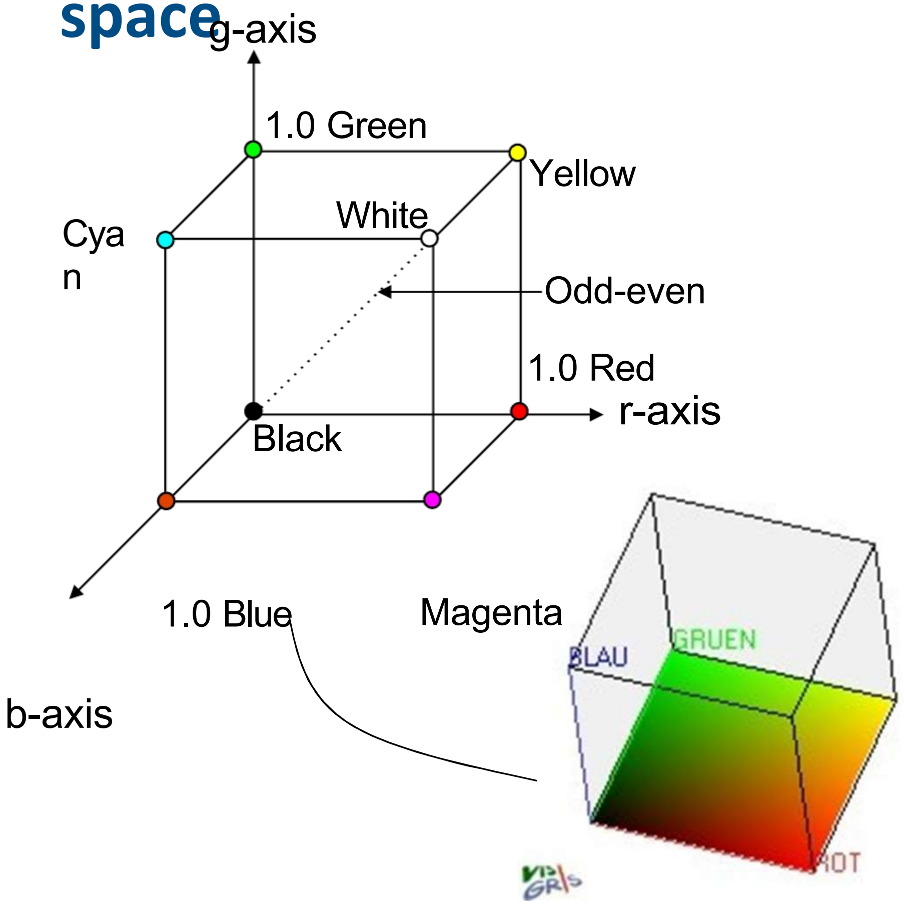
The RGB colour space



General:

- 3D colour space
- Coordinate values must be between 0 and 1.
- Colour on the surface of the cube or inside it
- Colour is described by a 3D vector:
  - Colour = $[r, g, b]^t$
  - Red = $[1, 0, 0]^t$
  - Origin: Black $[0, 0, 0]^t$
- lowest brightness: $[0, 0, 0]^t_{RGB}$
- **maximum brightness?**

The RGB colour space



Composition of a colour:

additive colour

mixture yellow= red

+ green

= $[1,0,0]^t + [0,1,0]^t$
= $[1,1,0]^t$

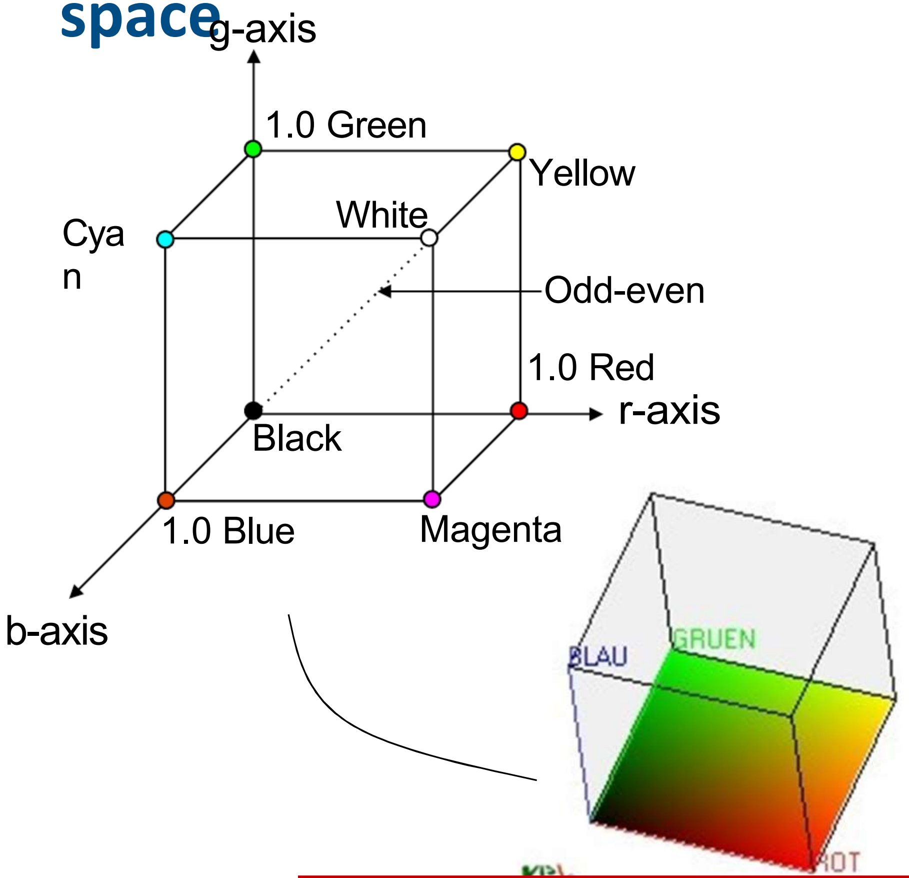
White= Red + Green + Blue
= $[1,0,0]^t + [0,1,0]^t + [0,0,1]^t$
= $[1,1,1]^t$

Yellow (medium light) = $[0.5,0.5,0]^t$

Cyan = $[0,1,1]^t$

Magenta = $[1,0,1]^t$

The RGB colour space



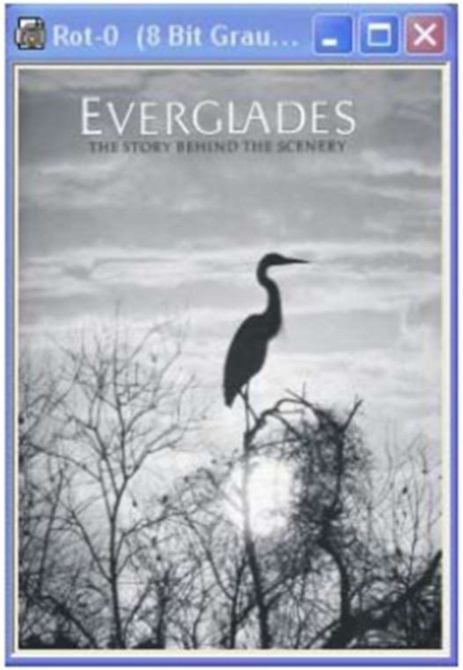
Properties of the RGB model

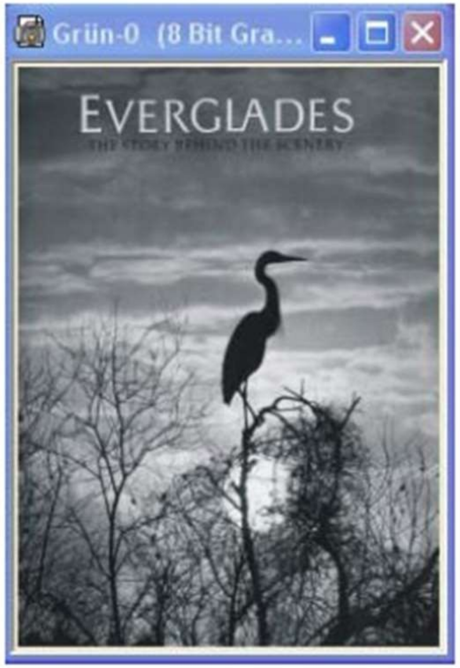
- Appearance of the colour is primarily determined by the largest component(s).
- If all colour components have the same value, it is a grey tone (**achromatic**).
- **Not** considered in the model:
  - Brightness differences in blue and green hues - see colour perception.

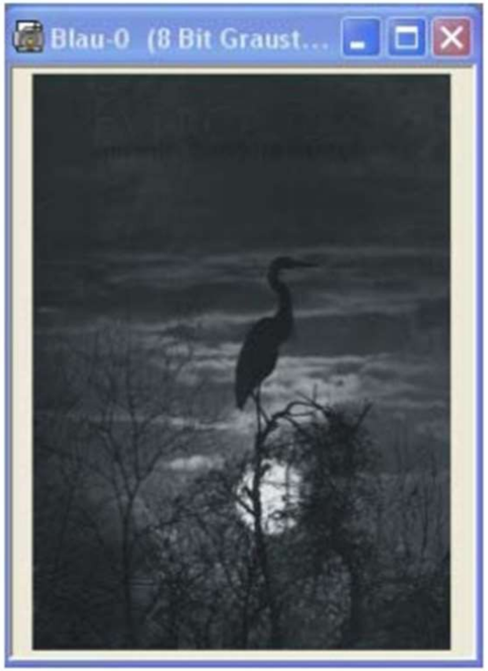
Exercise: RGB colour model

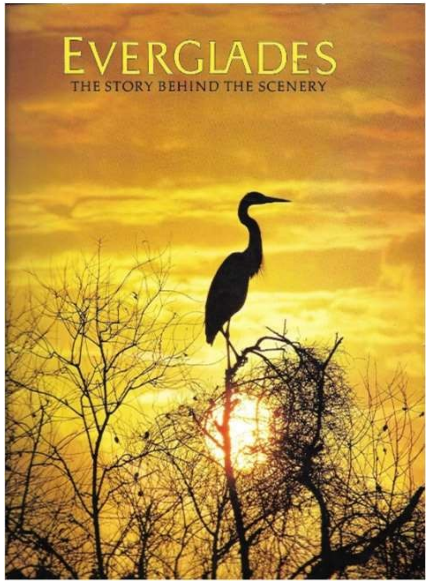
The image was split into a red, a green and a blue channel.

- What colour is the lettering "EVERGLADES"?
- What colours are the sky, the sun and the bird?









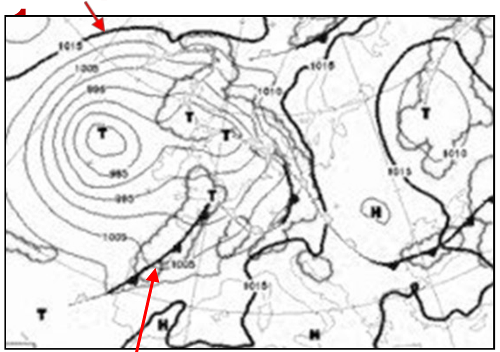
Exercise: RGB colour model

The image was split into a red, a green and a blue channel.

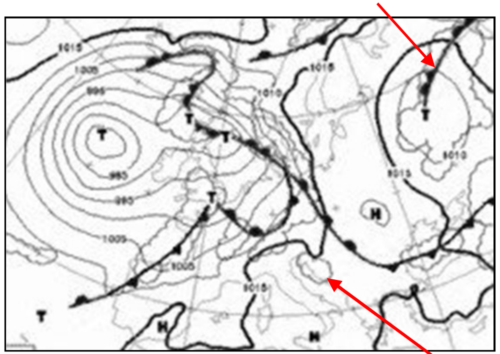
- What colours are the marked fronts and low pressure areas?

Low pressure area 1:

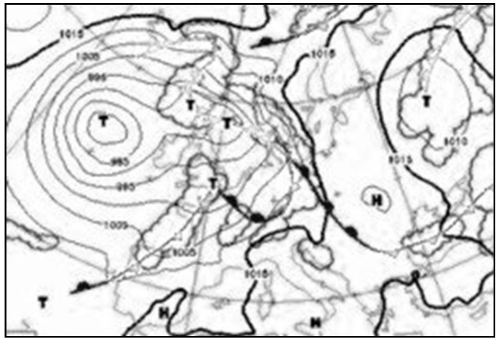
Front 2:



Red channel

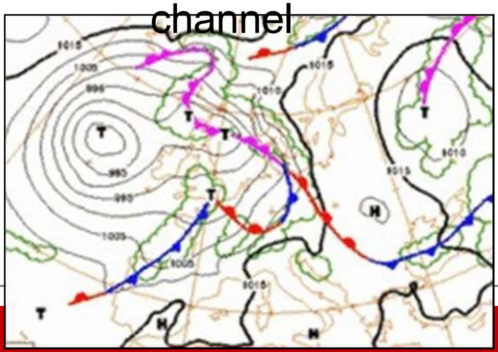


Green channel



Blue channel

Front 1:

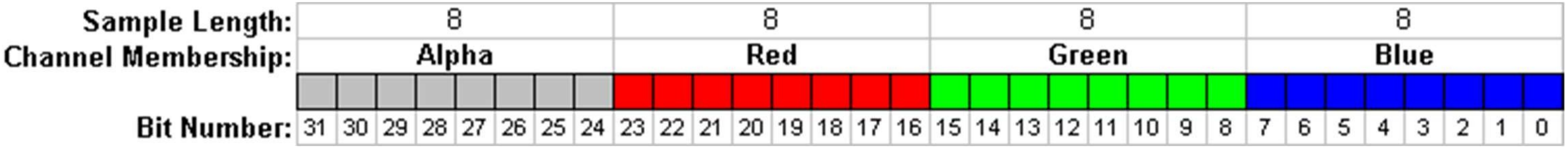


Original RGB image

Low pressure area 2:

Extension of the RGB system to RGBA

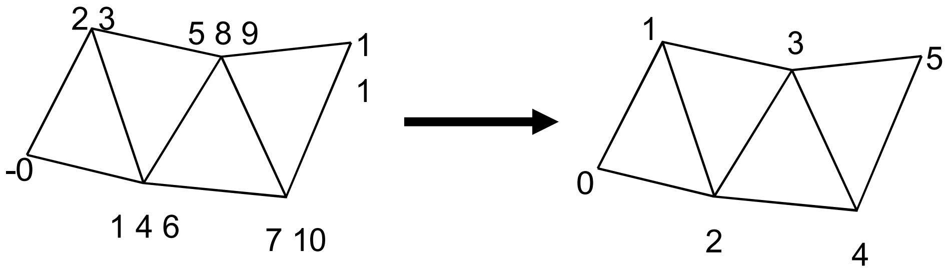
32 bit colour image:



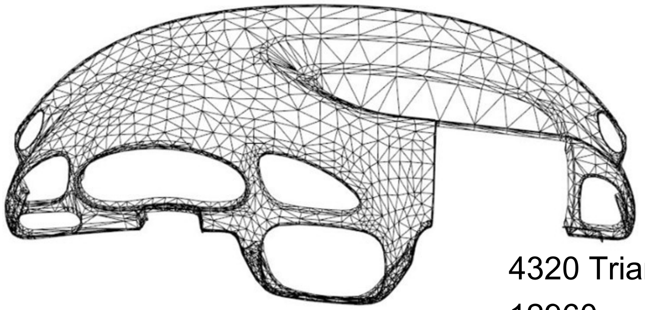
Sample Length:	8								8								8								8							
Channel Membership:	Alpha								Red								Green								Blue							
	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦
Bit Number:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Triangle Strips / Triangle Strips

- The aim is to create as few elements/vertices as possible.
- Corner points can be recycled through connected strips.



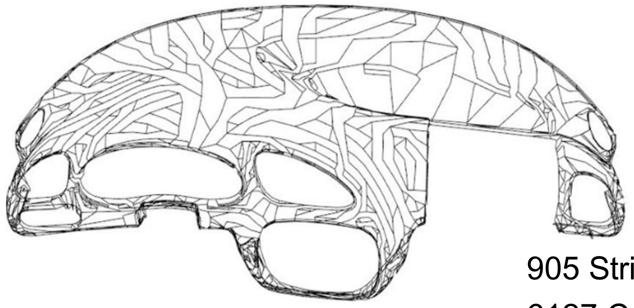
Number of points without streaking and with streaking



4320 Triangles

12960

Cornerstones

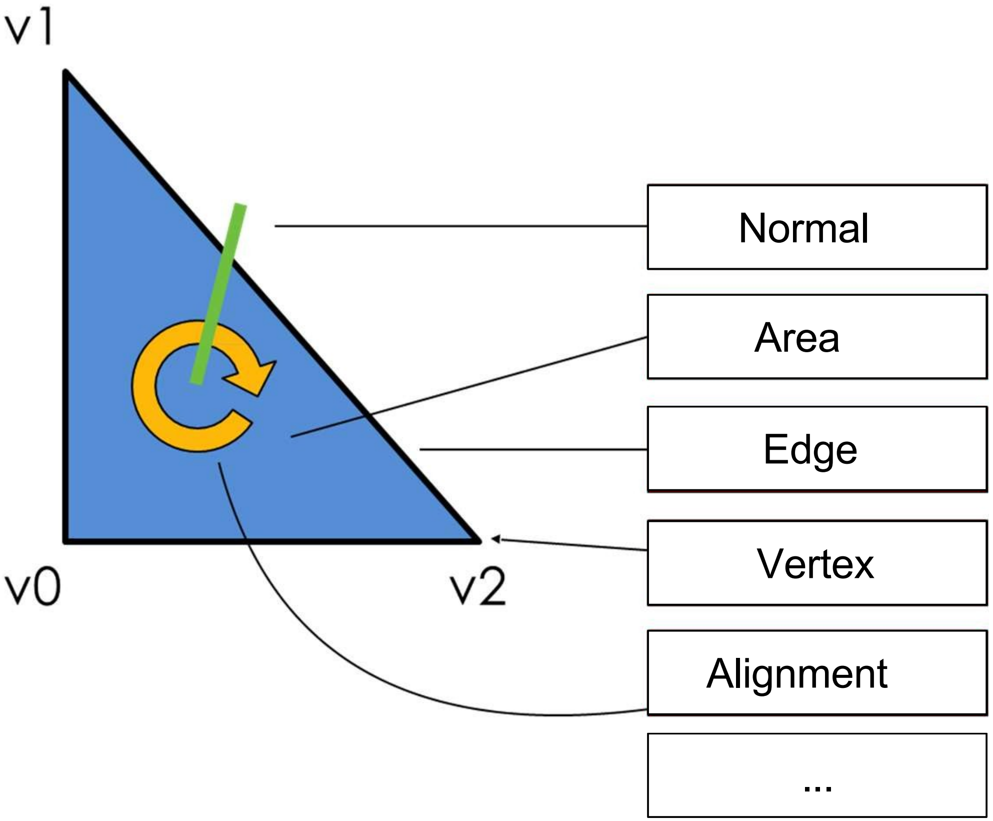


905 Strips

6127 Cornerstones

Scene description

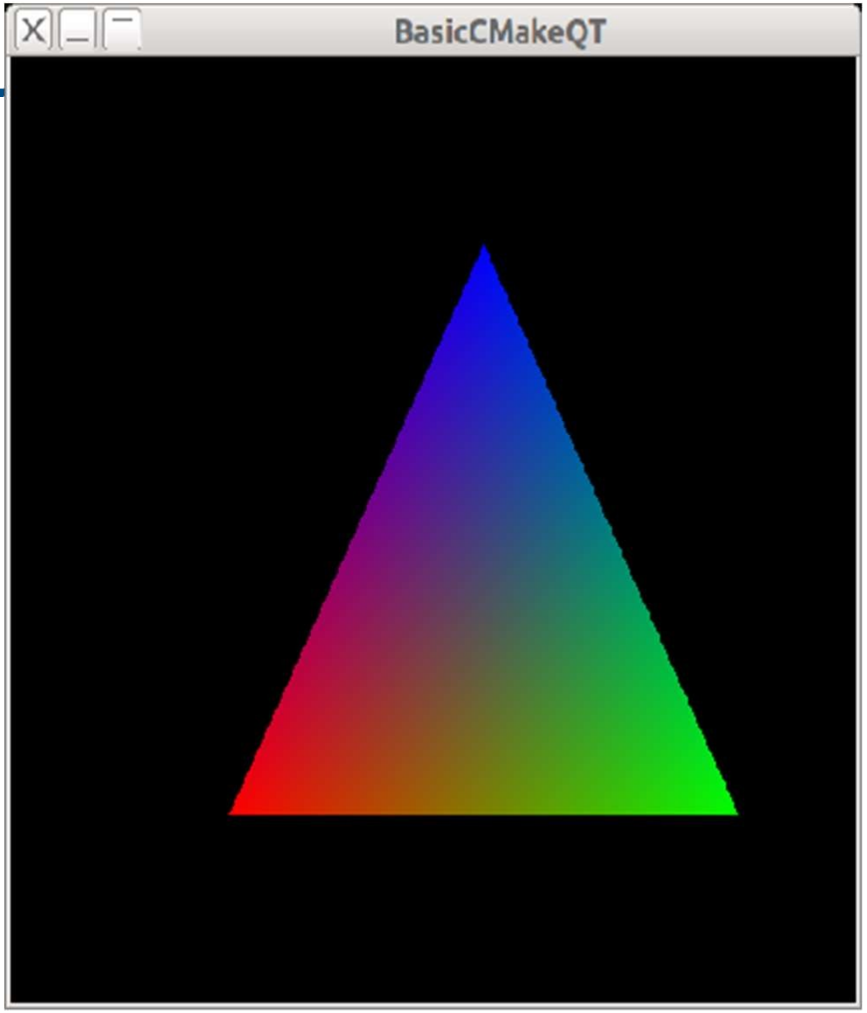
- What is a primitive? It consists of
  - Corner points/vertices
  - Edges
  - Areas
  - Alignment
  - Normal
- **But all this can be derived from the vertices!**



Scene description II

Example:

- Save vertices
- 2D coordinates and RGB colours
- (5D total)



CHAPTER 4

OpenGL

General information about OpenGL

- OpenGL (Open Graphics Library) is a specification of an API for 3D graphics
- OpenGL specifies (standardised) about 250 commands
- The implementation of the commands can be found in the graphics card drivers
  - Commands are then executed either by the graphics card
  - or on the CPU
- OpenGL is a rendering system, not a modelling software: complex models must be built from simple graphical primitives.

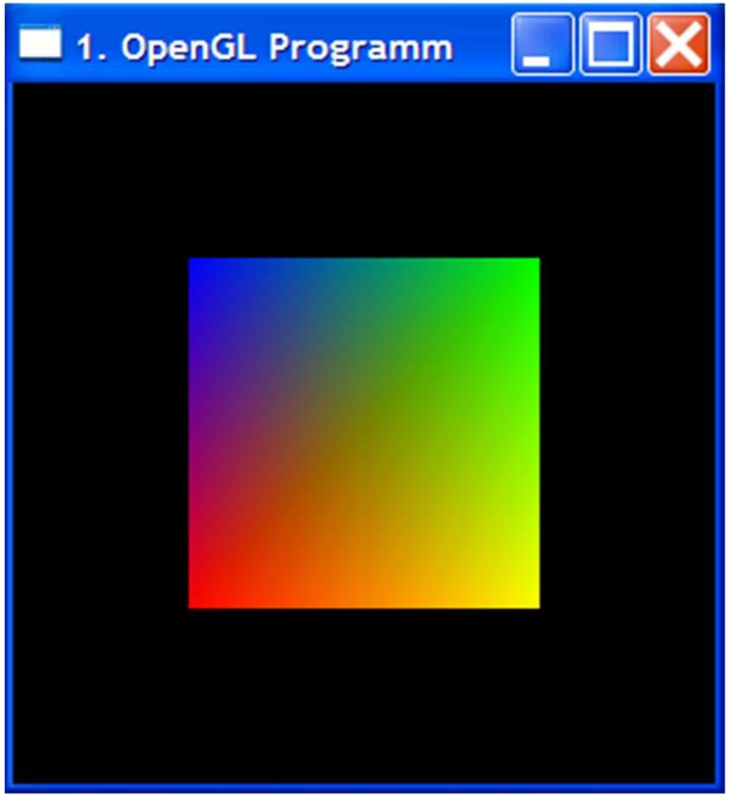
General information on OpenGL II

- OpenGL is a **state machine:**
  - Functions change the internal state or use it for representation.
  - This means that once switched on, the respective state remains active until it is switched off again or switched over.
- OpenGL is very "**explicit**":
  - What has not been explicitly activated remains off.
  - Example: It is of no use to set the transparency if you have not explicitly said that transparencies should be calculated.
- We use **GLFW** (Graphics Library Framework), an open-source, multiplatform Library for OpenGL
  - Plus GLEW as extension for certain additional functions
  - The only alternative: GLUT (OpenGL Utility Toolkit), or **FreeGlut** as a further development.

Legacy code (OpenGL 1.x)

```
glBegin( GL_POLYGON );

         // State-Machine: If only the first colour
            value is specified have
         // all following corner points the same
            colour value

         glColor4f( 1., 0., 0., 1.);
         glVertex3f( -0.5, -0.5, 0);

         glColor4f( 1., 1., 0., 1.); // Yellow
         glVertex3f( 0.5, -0.5, 0);

         glColor4f( 0., 1., 0., 1.);
         glVertex3f( 0.5, 0.5, 0);

         glColor4f( 0., 0., 1., 1.);
         glVertex3f( -0.5, 0.5, 0);

glEnd();
```

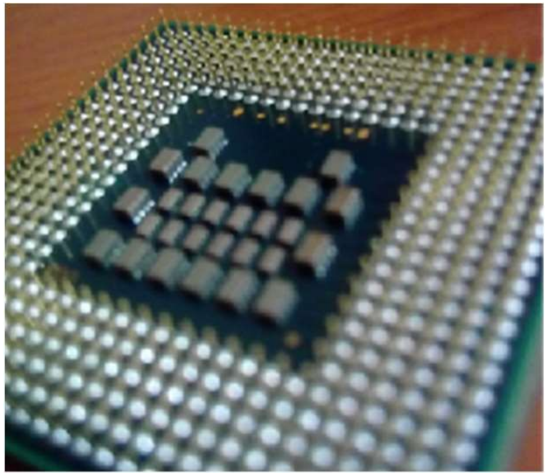
Output of the OpenGL programme:



Upload of data from CPU to GPU is very slow.

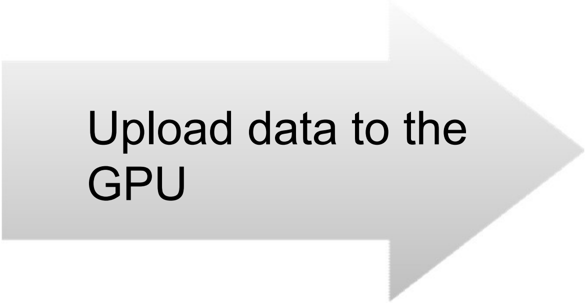
Interaction CPU and GPU

- **Today**: OpenGL offers functions that address both CPU and GPU

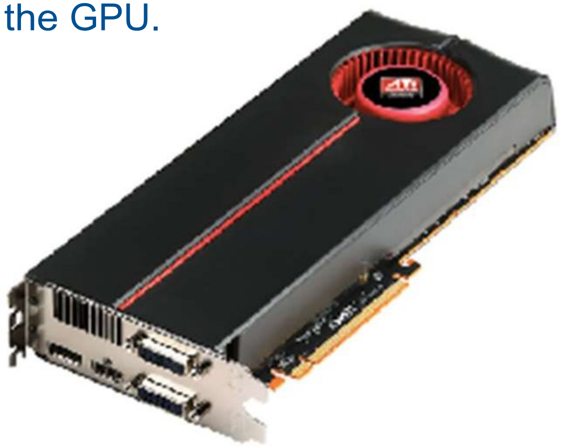


©Lamproslefteris

Goal: Upload a large amount of data to the GPU.
Speeds up the process immensely.



Upload data to the GPU



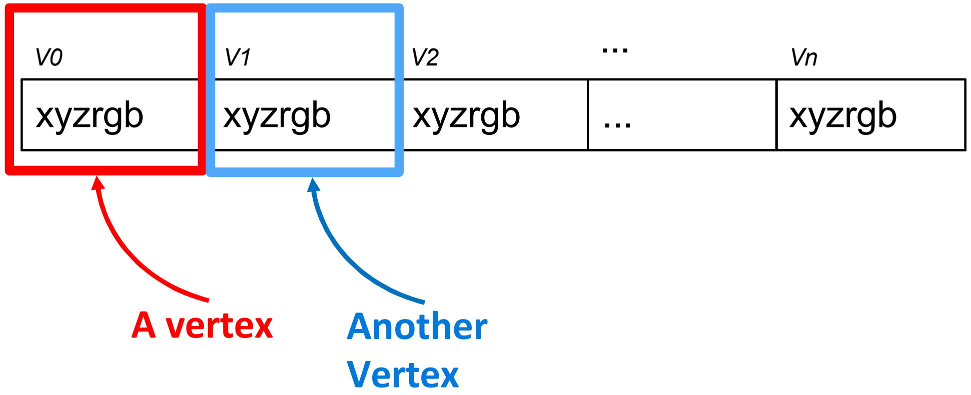
©Advanced Micro Devices (AMD)

- Data creation/manipulation
- Programme logic

- Data processing
- Image generation

Data creation

- The geometry is stored as a contiguous data array
  - **Interleaved**: All data of a Vertex $_{Vi}$ are in succession



V0	*V1*	*V2*	…	*Vn*
xyzrgb	xyzrgb	xyzrgb	...	xyzrgb

A vertex

Another Vertex

- In source code:

```
float vertices[] = {-0.5, -0.5, 0.0, 0.0, 1.0, // Here in 2D, xyrgb
                    0.5, -0.5, 0.0, 0.0, 1.0,
                    0.5, 0.5, 0.0, 1.0, 0.0,
                    0.0, 1.0, 1.0, 0.0, 0.0,
                    -0.5, 0.5, 0.0, 1.0, 0.0};
```

The Vertex Buffer Object

- The geometry data must be loaded into the memory of the GPU
  - For this purpose, OpenGL provides special memory buffers, the so-called "memory buffers". **Vertex Buffer Objects (VBO)**

- Objects on the GPU can be referenced via (unique) IDs

- The data transfer to the GPU (almost) always takes place in three steps:
  - Generate an ID
  - Activating the corresponding memory buffer (see "State Machine")
  - Upload the data

VBOs in OpenGL (overview)

Generate ID

- `void glGenBuffers(GLsizei n, GLuint * buffers);`
  - creates and returns one (or n) buffer ID, which is/are stored in `buffers`
  - `n:` Specifies the number of buffer objects to be generated, usually 1.

Activate correct buffer

- `void glBindBuffer(GLenum target, GLuint bufferID);`
  - `target:` Type of buffer to be described
    - `GL_ARRAY_BUFFER:` Buffer with the actual geometry data
    - `GL_ELEMENT_ARRAY_BUFFER:` Buffer with indices to another VBO
  - `bufferID:` The ID of the VBO previously generated with glGenBuffers.

Delete data

- `glDeleteBuffers(GLuint bufferID);`
  - Deleting a buffer with the ID `bufferID`

VBOs in OpenGL (overview)

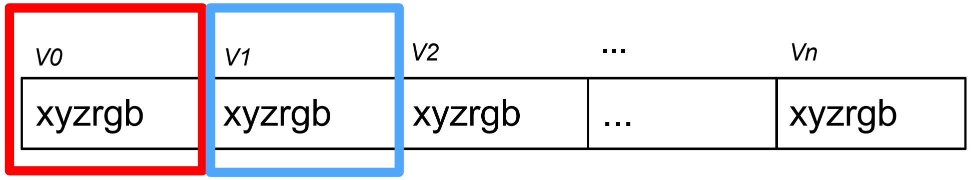
Uploading the data to the GPU

- `glBufferData(GLenum target, GLsizeiptr size, const void * data, GLenum usage);`
  - `target`: Type of buffer to be written to (as for `glBindBuffer`).
    - `GL_ARRAY_BUFFER, GL_ELEMENT_ARRAY_BUFFER, ...`
  - `size`: size of the data to be written **in bytes** (size of float = 4 bytes, RGB => 12 bytes)
  - `data`: pointer to the first element of the data to be uploaded
  - `usage:` type of (expected) later access to the data
    - `GL_STATIC_DRAW:` initialised once, rendered frequently
    - `GL_DYNAMIC_DRAW:` frequently changed and rendered
    - `GL_STREAM_DRAW:` rarely changed or rendered
    - …

VBOs in OpenGL

Question: What do the data in the VBOs look like?

- OpenGL is stupid! It only sees a bunch of allocated memory



VBO with vertex data

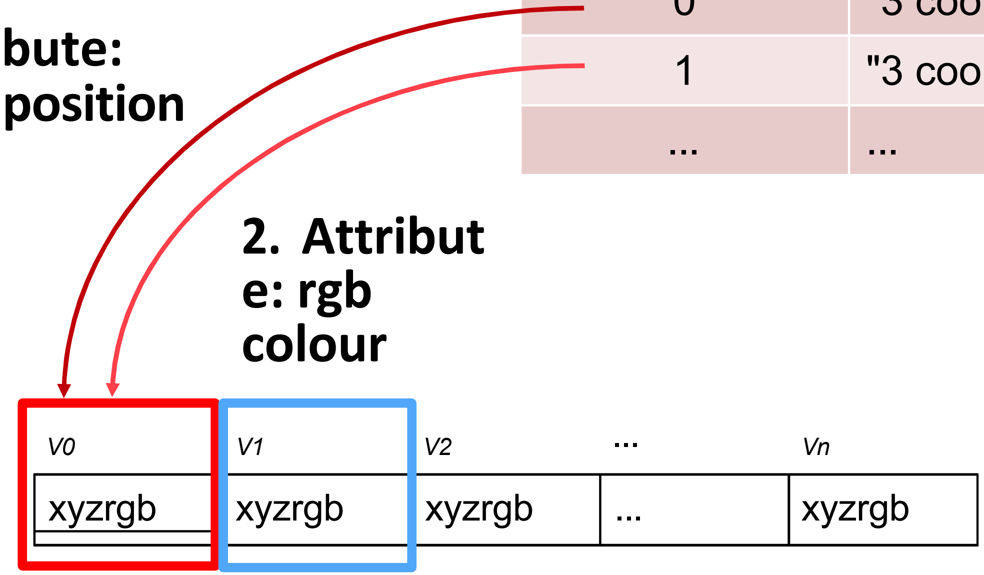
- An additional explanation is needed on how to interpret the data!
  - a **Vertex Attribute Object** is created for each object to be rendered
  - Pointers to the individual attributes are stored here (position, colour, etc.).

The Vertex Array Object (VAO)

Each attribute pointer describes a Vertex attribute

Attribute index	Description
0	"3 coordinates of the type GL_FLOAT per vertex".
1	"3 coordinates of the type GL_FLOAT per vertex".
...	...

1. Attribute: Vertex position



2. Attribut e: rgb colour

But beware: The interpretation (coordinates, colour etc.) will be determined later.

VAOs in OpenGL

Generate ID (analogue to VBO)

- `void glGenVertexArrays(GLsizei n, GLuint * arrays);`
  - creates and returns one (or n) array ID, which is/are stored in `buffers`
  - `n:` Specifies the number of buffer objects to be generated, usually 1.

Activate correct VAO

- `void glBindVertexArray(GLuint vaoID);`
  - `vaoIO:` The ID of the VAO previously generated with glGenVertexArrays.
  - Only one VAO can be active at a time!

Delete VAO

- `void glDeleteVertexArrays(GLuint vaoID);`
  - `vaoIO:` The ID of the VAO previously generated with glGenVertexArrays.

VAOs in OpenGL II

Define the vertex attributes (tell OpenGL where to find which data).

- `glVertexAttribPointer(GLuint index, GLuint size, GLenum type, GLboolean normalized, GLsizei stride, const void * offset);`
  - `index`: User-defined ID of the attribute (needed later on the GPU)
  - `size`: number of "coordinates" per vertex, usually 3
  - `type`: Data `type`, e.g. GL_SHORT, GL_INT, GL_FLOAT, GL_DOUBLE
  - `normalised`:
    - false: Floating point values are provided, no normalisation required
    - true: the provided values are mapped to the range [0,1] for unsigned data and [-1,1] for signed data
  - `stride`: Size of a vertex **in bytes**.
  - `offset`: memory offset **in bytes at** which the vertex attribute in the active array begins (0 for the first attribute)
- glVertexAttribPointer always refers to the **currently active VBO (and VAO) !**

VAOs in OpenGL III

Switching individual attributes on and off

- `glEnableVertexAttribArray(GLuint index);`
- `glDisableVertexAttribArray(GLuint index);`
  - The index corresponds to the self-selected index in

    `glVertexAttribPointer(GLuint index, ...)`

- By default, all attributes are deactivated!

- Application example:
  - Two render modes: one for visualising the position (e.g. for troubleshooting) and one for pure colour display can be combined in the same VAO

Code example

Create VBO for position and colour

```
float[] vertices = ... // stores the data of your vertices
// in this example 3 float for position followed by 3 floats for colour
GLuint vaoID, vboID;

// generate and activate VBO and upload data //
glGenBuffers(1, &vboID);
glBindBuffer(GL_ARRAY_BUFFER, vboID);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), &vertices, GL_STATIC_DRAW);

// generate and activate VAO //
glGenVertexArrays(1, &vaoID);
glBindVertexArray(vaoID);

// describe VBO in the VAO //
glVertexAttribPointer(0, 3, GL_FLOAT, false, 24, 0);
glEnableVertexAttribArray(0);

glVertexAttribPointer(1, 3, GL_FLOAT, false, 24, (void*)12);
glEnableVertexAttribArray(1);
```

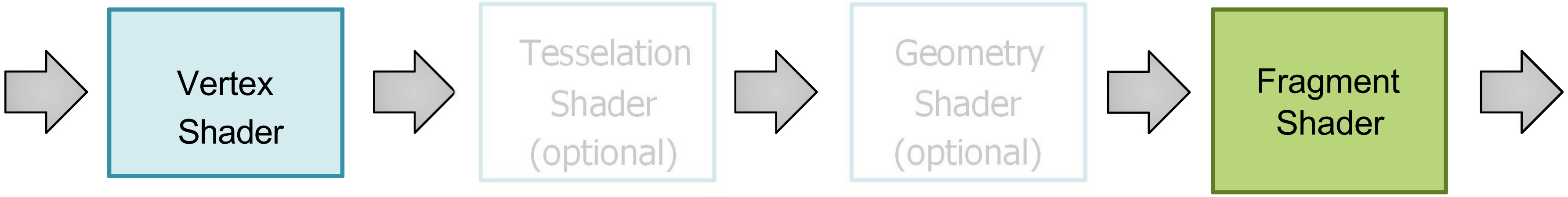
Two attribute pointers are defined, the vertex positions are given the ID 0 and the colours the ID 1 in the VAO.

Note the offset and the strange data type!

Data processing

- The uploaded data can still be manipulated before rendering.
- **Advantage:** The calculations are carried out in parallel per vertex or per pixel.

- The desired calculations are implemented in so-called **shaders**, small programme fragments that can be executed directly on the GPU.



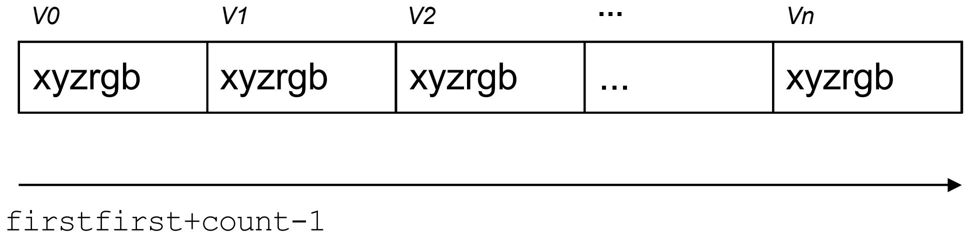
- Each shader has a precisely defined task:
  - **Vertex Shader:** Calculates the **final position of** each vertex in the output image.
  - **Fragment Shader:** Calculates the **colour of each pixel** in the output image

Image generation

Rendering of a Vertex Array Object (and all connected VBOs)

- `glDrawArrays(GLenum mode, GLint first, GLsizei count);`
  - Goes through the entire VAO, starts at the primitive `first` and draws the first `first + count – 1` primitive
  - `mode`: type of primitives to be drawn, e.g. `GL_TRIANGLES`

- Inflexible, but fast

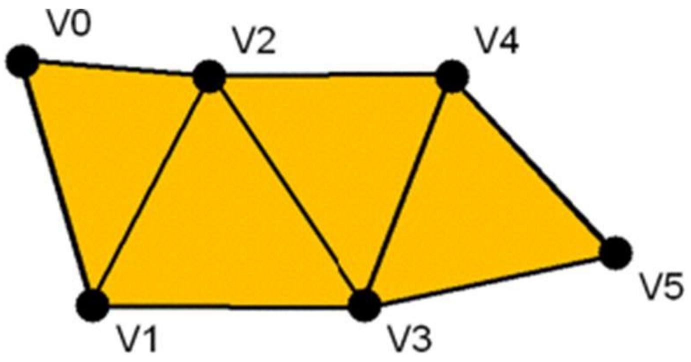


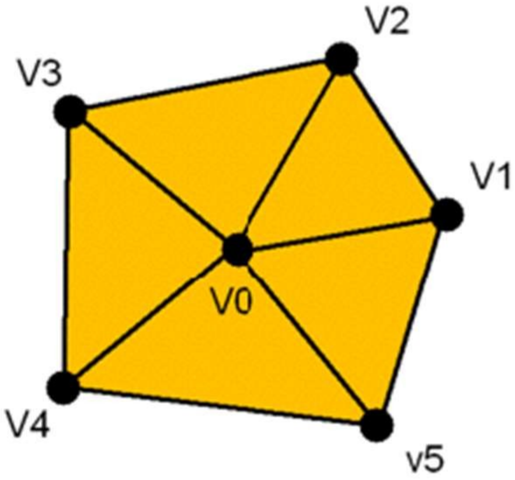
V0 *V1* *V2* ··· *Vn*

xyzrgb	xyzrgb	xyzrgb	...	xyzrgb

firstfirst+count-1

Index Buffer Objects





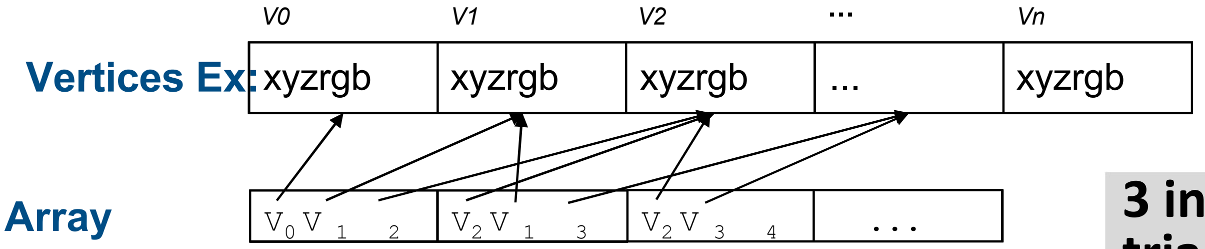
- Vertices are often shared by triangles (cf. slide *Triangle Strips*)
- Reuse of vertices through indexing
- Additional element array buffer required

- `glGenBuffers(1, &iboID);`
- `glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, iboID);`
- `glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(data), &data, GL_STATIC_DRAW);`

Index Buffer Objects II

Dereferencing via indices

- `glDrawElements(GLenum mode, GLsizei count, GLenum type, const void* indices);`
  - `mode:` drawing primitive, e.g. `GL_TRIANGLES`
  - `count:` Number of elements (i.e. indices, not complete triangles! □ Size of the indices array)
  - `type:` Type of data in the indices, `GL_UNSIGNED_BYTE, GL_UNSIGNED_SHORT, GL_UNSIGNED_INT`
  - `indices:` Pointer to the beginning of the active index array, normally 0.



Vertices Ex:

V0	V1	V2	…	Vn
xyzrgb	xyzrgb	xyzrgb	...	xyzrgb

Index Array

V_0 V 1 2	V_2 V 1 3	V_2 V 3 4	. . .

3 indices per triangle!

Index Buffer Objects III

Difference between glDrawArrays() and glDrawElements()

- `glDrawArrays():` Brute-force approach

- `glDrawElements():` More flexible (and sometimes faster)
  - Reuse of vertices for reduced data transfer
  - Adaptation of the IBO makes it possible to render only parts of an object

Code example with IBO

Create VBO for position, with an Index Buffer Object

```
// given: array of vertices and index array //
float[] vertices = ...
int[] indices = ...
GLuint vaoID, vboID;

// setup VBO //
glGenBuffers(1, &vboID);
glBindBuffer(GL_ARRAY_BUFFER, vboID);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), &vertices, GL_STATIC_DRAW);

// setup VAO //
glGenVertexArrays(1, &vaoID);
glBindVertexArray(vaoID);
glVertexAttribPointer(0, 3, GL_FLOAT, false, 12, 0);
glEnableVertexAttribArray(0);

// setup IBO //
GLuint iboID;
glGenBuffers(1, &iboID);    //only works after glGenVertexArrays();
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, iboID);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices, GL_STATIC_DRAW);
```

Code example with IBO

Rendering with IBO

```
void setup(){
         // given: array of vertices and index array //
         // setup VAO //
         // setup VBO //
         // setup IBO //
}

void render(){
         // activate VAO //
         glBindVertexArray(vaoID);
         // render call //
         glDrawElements(GL_TRIANGLES, count, GL_UNSIGNED_INT, 0);

         // good programmers should reset //
         glBindVertexArray(0);
}
```

VBOs are implicitly activated

Number of indices, not primitives!

Specify primitives and data type (of the IBO)