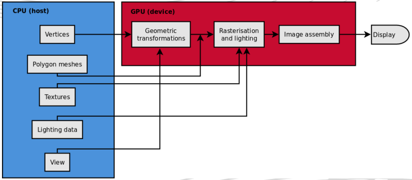
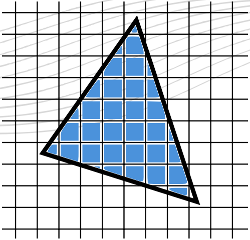
# Introduction
# and your first triangle

## Computer Graphics (DT3025)

Martin Magnusson

October 31, 2016



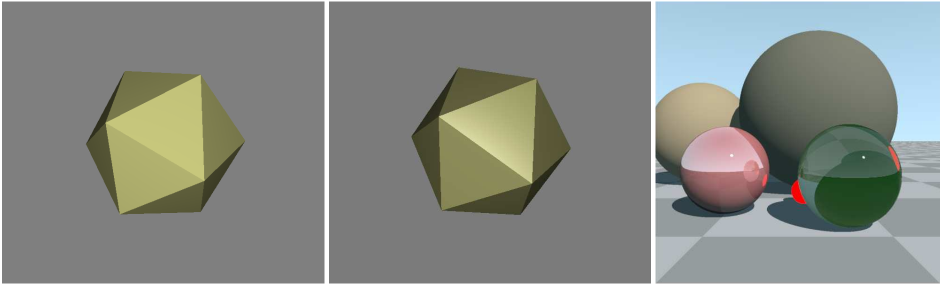



# What is computer graphics?

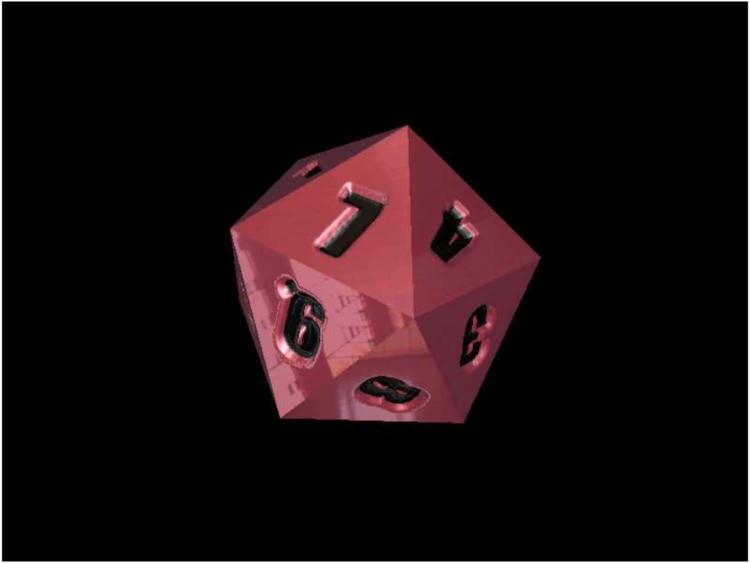
Visually communicate data… for lots of applications!


*The Last of Us*

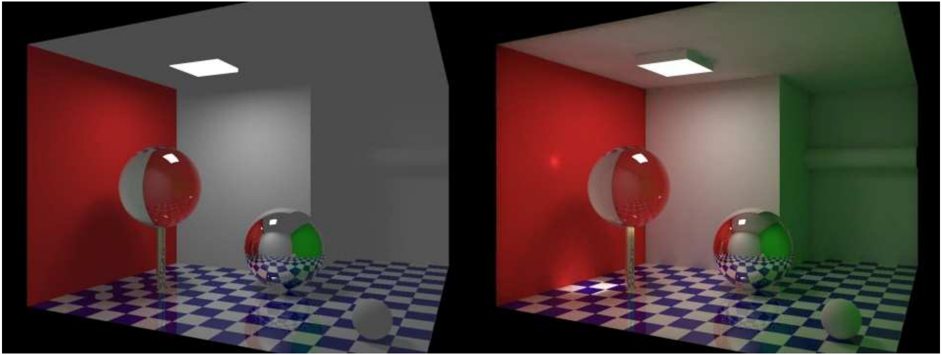
# Light and material models



# Textures and other mapping techniques

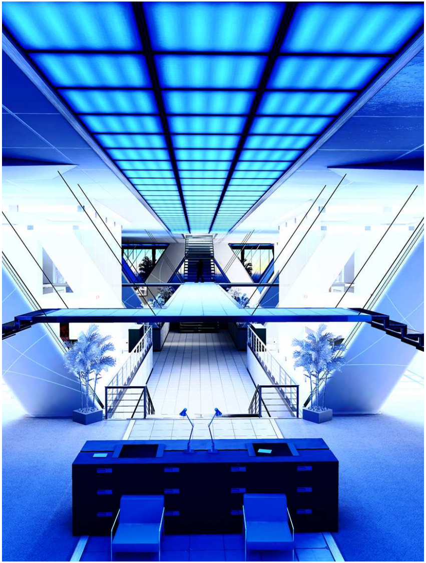


# Ray tracing and global illumination



# Game techniques for shadows and realistic lighting

*Doom 3: id software*



*Mirror's edge*

# And the math required to do it

$$L^{\mathrm{e}}(P,\boldsymbol{\omega}_{\mathrm{o}}) + \int_{\boldsymbol{\omega}_{\mathrm{i}}} L(R(P,\boldsymbol{\omega}_{\mathrm{i}}), -\boldsymbol{\omega}_{\mathrm{i}}) f_{\mathrm{s}}(P,\boldsymbol{\omega}_{\mathrm{i}},\boldsymbol{\omega}_{\mathrm{o}})(\boldsymbol{\omega}_{\mathrm{i}} \cdot \mathbf{n}_P)\,\mathrm{d}\boldsymbol{\omega}_{\mathrm{i}}$$

$$\cdots \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdots$$

# Teachers

**Lectures** Martin Magnusson: martin.magnusson@oru.se, room T-1216

**Labs** Daniel Canelhas: daniel.canelhas@oru.se, room T-1119

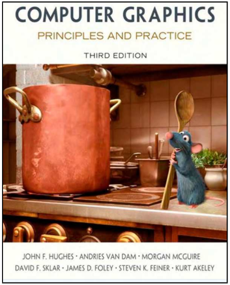
# Lecture schedule

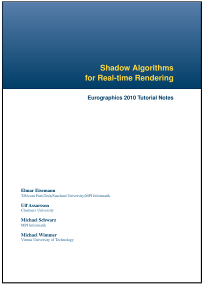
| | |
|---|---|
| today | Intro, and how to draw a triangle |
| Tue, 1 Nov | Linear algebra and perspective |
| Mon, 7 Nov | Materials and lighting |
| Tue, 8 Nov | Textures and mapping techniques |
| Mon, 14 Nov | Rasterised shadows |
| Tue, 15 Nov | Ray casting and ray tracing |
| Mon, 21 Nov | Ray tracing and path tracing |
| Mon, 28 Nov | Path tracing |
| Mon, 5 Dec | (placeholder) |
| Mon, 12 Dec | Case studies, summary |
| Thu, 12 Jan | Exam |

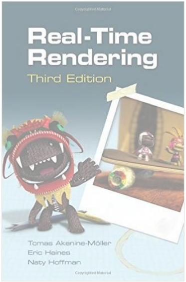
*Any changes to the schedule will be announced on Blackboard.*

# Course literature: theory

- **Hughes et al.**, *Computer Graphics: Principles and Practice, 3rd ed.*
  - Main book.
- **Eisemann et al.**, *Shadow Algorithms for Real-time Rendering*
  - For the lecture on rasterised shadows.
- (**Akenine-Möller et al.**, *Real-Time Rendering, 3rd ed.*)
  - Good alternative (e-)book. Not required.
- Don't forget: lecture contents (keep notes!), and the slides. (But the slides cannot contain all of the course contents.)

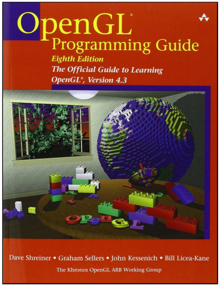


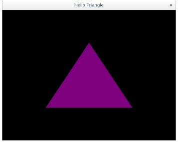
# Course literature: practice

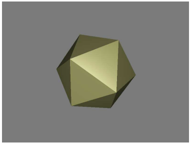
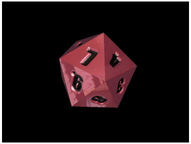
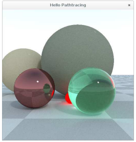
- OpenGL wiki: www.opengl.org/wiki/Main_Page
- OpenGL API documentation: https://www.opengl.org/sdk/docs/man4/
- **Shreiner et al.**, *OpenGL programming guide, 8th ed.*
  - Good reference (for OpenGL 4.3), but the web resources are enough.

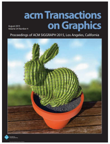


# Lab assignments

1. Intro to OpenGL, transformations
2. Materials and lighting
3. Textures and other mapping techniques
4. Ray tracing, path tracing
5. State-of-the-art literature review





# Course objectives

- Knowledge and comprehension:
  - Explain theory and methods from computer graphics.
    - *E g, rasterisation, path tracing, material models, shadow techniques.*
- Proficiency and ability:
  - Demonstrate skills in applying theory and methods for programming 3D graphics with industrially relevant methods.
    - *E g, rendering and shader programming with OpenGL.*
- Judgement and approach:
  - Demonstrate the ability to continuously update knowledge within the field of computer graphics as the state of the art progresses.
    - *Review current literature in Lab 5.*

See the full course plan on the web.

# Today's lecture

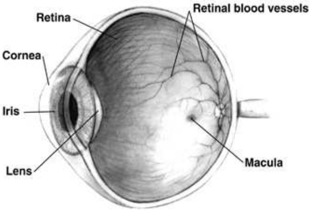
- Vision and perception.
- Computer graphics hardware.
- Your first triangle in OpenGL.

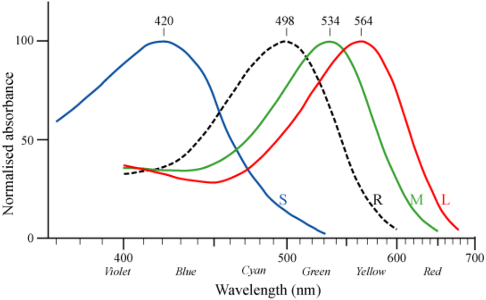
## Reading material

- Hughes et al.:
  - Chapter 1,
  - Chapter 5,
  - Chapter 15.7,
  - Chapter 16.1,3,
  - Chapter 28.1–3,5–8.
  - (Chapter 38 can be skimmed too.)

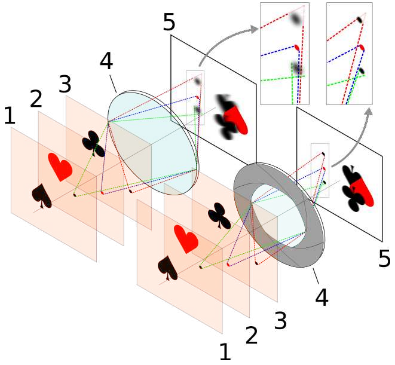
# What do we see?

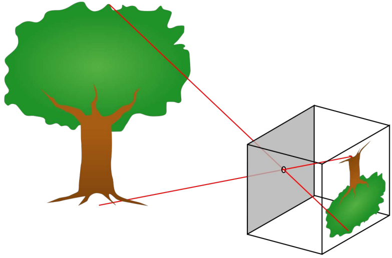
# How does the eye work?



# Assume pin-hole camera



Wikimedia Commons

# Assume geometric optics

Then we can’t do:

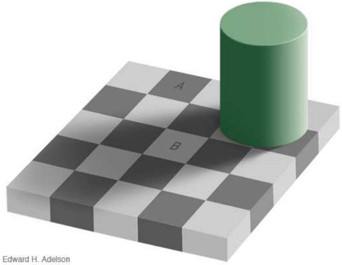
*physics.mun.ca*

*sunglasses-shop.co.uk*

*Ingrid Moen et al.*

# What is colour?

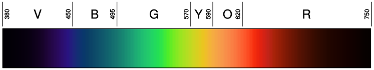
- Colour is a *sensation*.
  - *Influenced* by light's mix of wavelengths, but not determined by it.
  - Example: different physical inputs can be perceived very similarly, and vice versa.




  - Example: ”all cats are grey in the dark”
- Consider that about 5% of northern-Europeans are partially colourblind (8–10% of men, 1% of women).

# Reproducing colours

- Colour spectrum continuous.



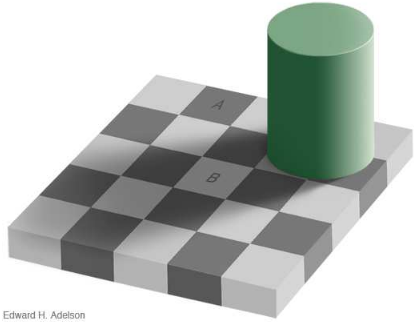

- (Sub-)pixels can only have one colour.
- What to do?
  - Model colours as weighted combination of basis colours.
    - RGB for screens (additive)
    - CMY for paper (subtractive)


*Spigget, via Wikipedia*

# Colour fundamentals

**Brightness** Perceived intensity (subjective)

**Luminance** (objective) measure of amount of energy that an observer receives

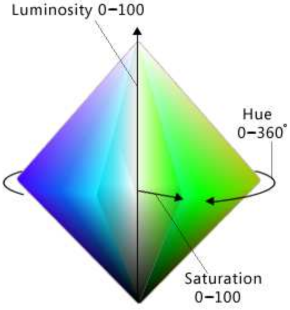


# Colour fundamentals

**Hue** Associated with dominant wavelength

**Saturation** Relative colour purity (how little white light mixed in)

**Chromacity** Hue and saturation together



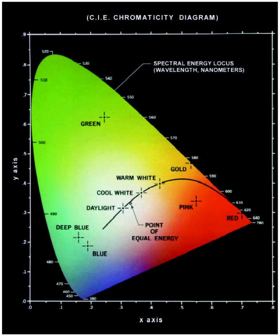

MSDN

# Colour fundamentals

**Achromatic** Uncoloured (white = unsaturated)

**Chromatic** With colour

**Monochromatic** (Almost) only one wave length (laser)




*General Electric co., via Gonzales et al. 2007*

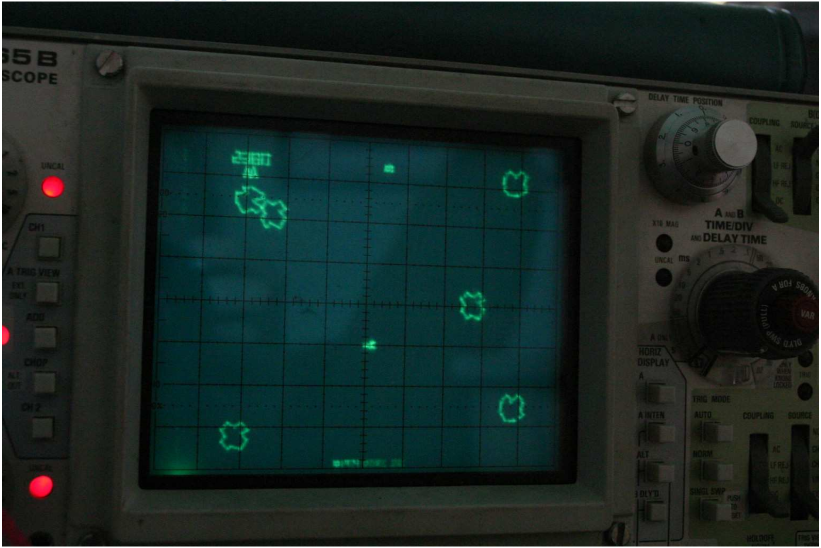
# Colour space and gamut

- Full CIE colour space is *tristimulus* model: $x, y, z$.
- Projection of $xyz$ space where $x + y + z = 1$.
- Shows gamut of "standard person".
- Monochromatic colours along edge (full saturation).
- sRGB gamut overlayed.


*Spigget, via Wikipedia*

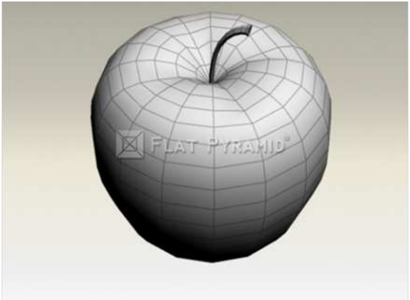
# Raster (bit-map) and vector graphics

# Actual vector graphics

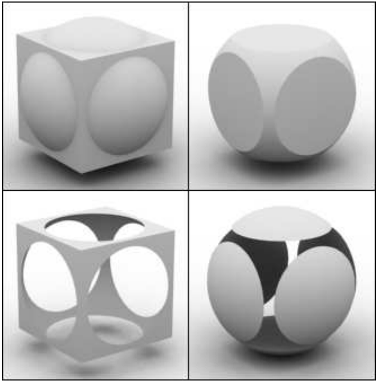


# 3D models

## Tesselated




## Procedural

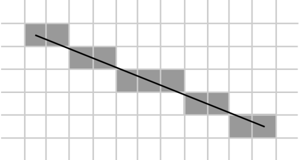


http://giuliapianacaad.altervista.org

# Rasterisation (lines)

How to draw vector graphics on a rasterised screen (or printer)?

- Draw primitives on an output matrix (raster).
- Bresenham's algorithm (lines):
  - Line defined by end points $(x_0, y_0), (x_1, y_1)$.
  - Can express it as $y = kx + m$.
  - Select which octant.
  - Start from top left $(x_0, y_0)$.
  - At each step increment $x$ and decide whether the next point should be $(x + 1, y)$ or $(x + 1, y + 1)$.
  - Compute error $e$ between pixel grid's $y$ coordinate and the line's $y$ coordinate.
  - Choose $(x + 1, y)$ if $e < 0.5$, and $(x + 1, y + 1)$ otherwise.



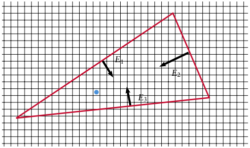
*Wikimedia commons*

# Rasterisation (triangles)

- Each triangle edge is on an infinite line.
- Line defines halfspace (inside vs outside).
- Triangle interior is all points inside the 3 halfspaces of its 3 edges.

$$E_i(x, y) = a_i x + b_i y + c_i$$

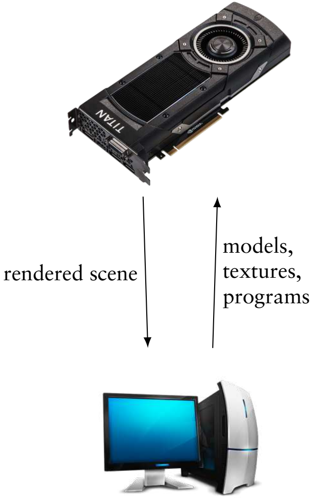
$$(x, y) \text{ within triangle } \equiv E_i(x, y) \geq 0, \forall i = 1, 2, 3$$




- Brute-force rasteriser:
- For each triangle
  - Compute $E_{1,2,3}$ from vertices (yields $a_i, b_i, c_i$)
  - For each pixel $(x, y)$
    - Evaluate edge functions (at pixel centre).
    - If all are $\geq 0$, pixel is inside.
- Is there a problem with that?
- Didn't I just say polygon objects have no inside?

# What is a GPU?

- One or more processors specialised on
  - parallel floating point operations,
  - vector and matrix computations ($4 \times 4$),
  - textures as matrices,
  - interpolation, blending, etc.
- Runs on-board *graphics cards*.
  - Limited access to memory.
  - Compile and upload program from CPU.
- We need an API for device-independent access to HW.



# History of graphics hardware

| | |
|---|---|
| **1960's** | Screens replacing printers as output |
| **1987** | VGA ($640 \times 480$ px, 256 col) |
| **1989** | SVGA ($1024 \times 768$ px, 256 col) |
| **1995** | First combined 2D/3D card<br>■ Fixed-function pipeline |
| **1997** | 3DFX "Voodoo" cards<br>■ Mip-mapping and Z-buffer in HW,<br>■ AGP replaces PCI |
| **1999–2002** | nVidia GeForce, 32–128 MiB RAM |
| **2008** | nVidia/ATi GeForce/Radeon, 256–1024 MiB RAM<br>■ Programmable shaders |

# Modern graphics cards


geforce.com

- 8000 million transistors
    - Intel Core i7 has ≈ 1000 million
- >3000 cores
    - Top consumer CPUs have 16
- 12 GiB memory
- >300 GiB/s memory bandwidth
    - Fast DRAM is around 20 GiB/s
- Polygon counts no longer important: more computations *per pixel*

# Graphics APIs

How to interface with hardware?

OpenGL

- All desktops/laptops

WebGL, OpenGL|ES

- All devices

Direct3D

- Windows only

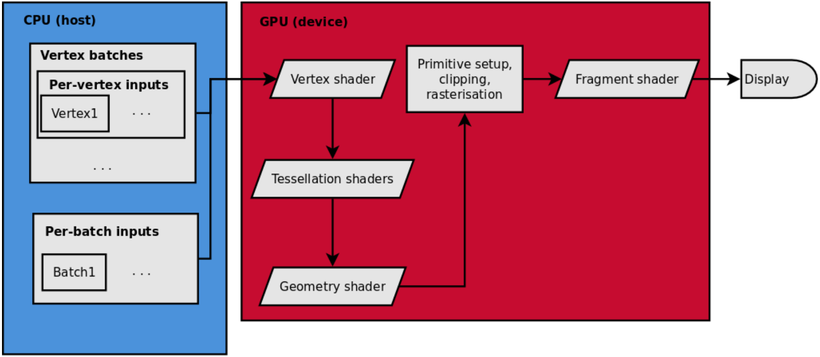
Vulkan

- “Next generation” OpenGL, allows more direct access to hardware (for better and worse)

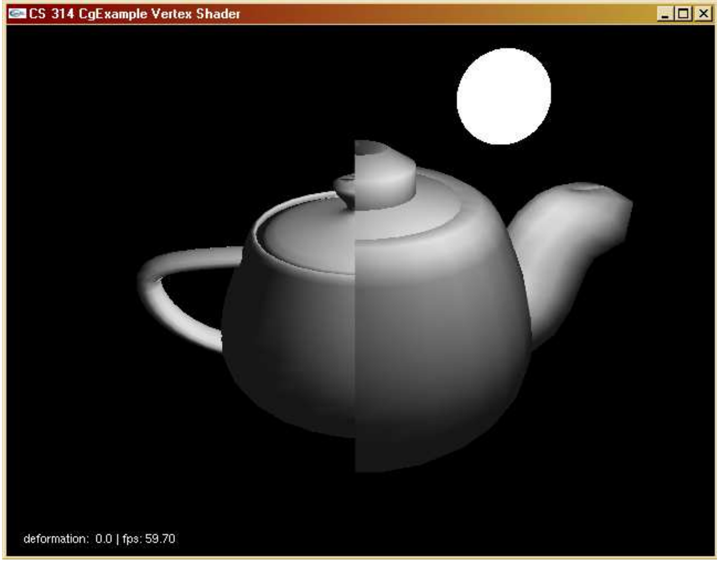
# Conceptual graphics pipeline: pseudocode

```
for (each triangle)
{
   transform into eye space;
   set up 3 edge equations;
   for (each pixel x,y)
   {
      if (it passes all edge equations)
         compute z
         if (z < zbuffer[x,y])
         {
            zbuffer[x,y] = z
            framebuffer[x,y] = shade()
         }
   }
}
```

# The programmable graphics pipeline



# What’s a vertex shader?

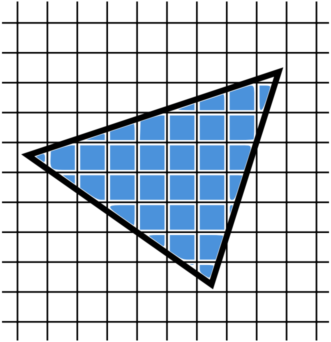



Left: trivial vertex shader. Right: also move each vertex in normal direction.

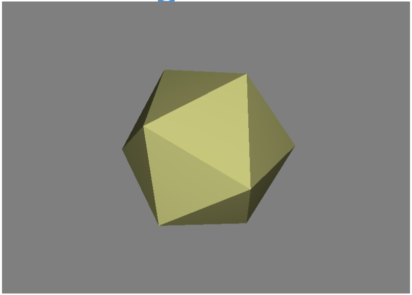
Gordon Wetzstein

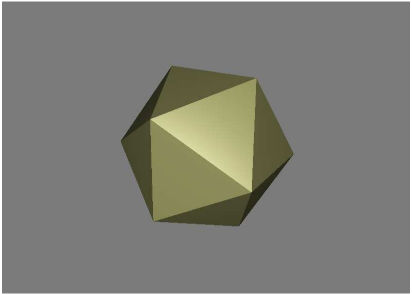
# What's a fragment?

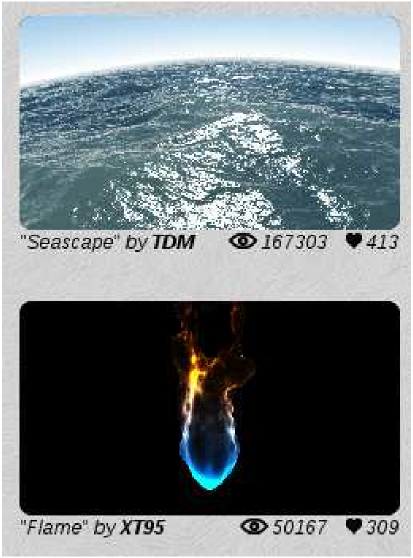
- *Pixel*: contents of frame buffer at a specific location (colour, depth, etc.)
- *Fragment*: state required to potentially update a pixel.
- Think of it as a *potential pixel*.
- If a fragment passes rasterisation tests, it will update a pixel in the frame buffer.



# What’s a fragment shader?







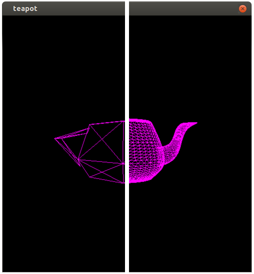

shadertoy.com

# Optional shaders

Additional shader stages between vertex and fragment shader.

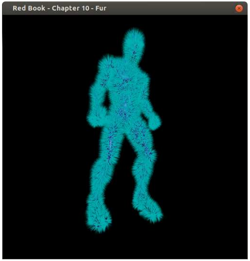
## Tessellation shaders

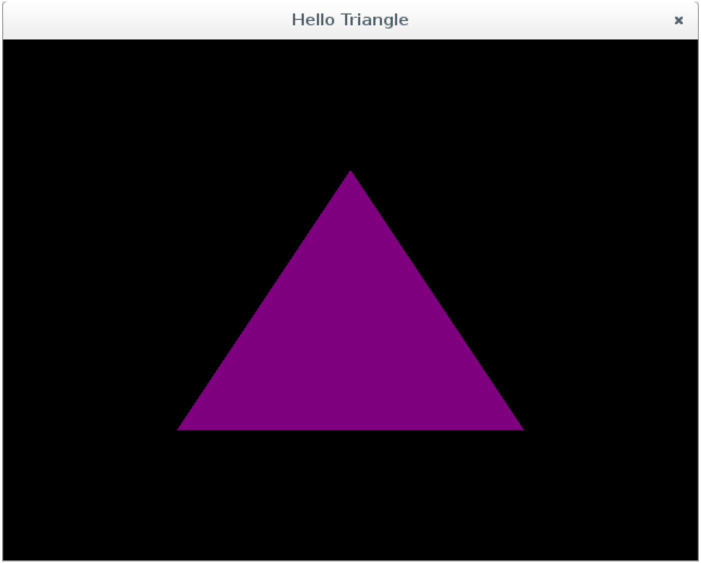
- Input:
  - (high-level) surface description
- Output:
  - Triangle list
- E g, adaptive subdivision of surfaces, or *displacement mapping*.




## Geometry shader

- Input:
  - A *primitive* (e g, a triangle)
- Output:
  - Zero or more primitives.
- E g, generate fur, render primitive on several images (for special effects)...




Hello Triangle

# How to draw a triangle, in six steps

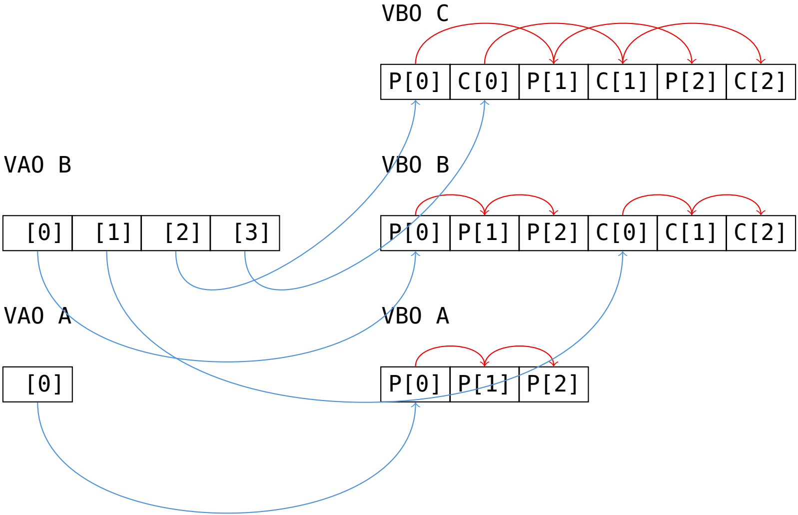
1. Create a *context* (a screen area) in which to draw.
2. Create a *vertex array object* (VAO) which will hold pointers to what to draw.
3. Create an array of *vertex attributes* (this is your triangle).
   - position, colour,
     - and/or normal, temperature... whatever you want
4. Put them in a *vertex buffer object* (VBO) and send them to GPU.
5. Compile a *shader program* to tell GPU what to do with the data.
   - Compile several shader *stages*, link them, and upload them.
6. Draw!

# First step: giving OpenGL somewhere to draw

## Excerpt: setting up rendering context in GLFW

```
// Init the GLFW helper library
if (not glfwInit())
  exit(EXIT_FAILURE);

// Tell the OS to create a window for us
GLFWwindow* window = glfwCreateWindow( 20, 20, "Title", NULL, NULL );

if (not window) {
      glfwTerminate();
      exit(EXIT_FAILURE);
}

glfwMakeContextCurrent( window );

// Now you can tell OpenGL what to draw in the window
```

# Buffer objects



# Draw a triangle: CPU-side

```
// Allocate 3 vertices with 3 attributes each (3D position):
float vertices[3][3] = {
  {-1.0,  -1.0, 0.0},
  { 1.0 , -1.0, 0.0},
  { 1.0 , +1.0, 0.0}};

// (Set up buffer objects)    . . .

// Specify how this attribute is stored:
glVertexAttribPointer(
  0,                        // attribute nr
  3,                        // size of attribute
  GL_FLOAT,                 // data type
  GL_FALSE,                 // normalise or not
  0,                        // stride
  (void*) &vertices);       // array offset

// (Set up shader program)    . . .
// (Enable and send data to GPU)    . . .

// Draw triangle primitives, using 3 points per primitive.
glDrawArrays(GL_TRIANGLES, 0, 3);
```

# Draw a triangle: vertex shader

- Vertex attributes are inputs to a *vertex shader* – executed once per vertex.
- Vertex shader outputs variables that are passed to rasterisation and fragment shader.
  - `gl_Position` determines the screen coordinates. Stretches from $(-1,-1)$ to $(+1,+1)$.
  - `out` variables are passed to fragment shader (after linear interpolation!)

## Shader code

```
layout(location=0) in vec4 inPosition; // Attribute nr 0 is position.
out vec3 myColour;

void main() {
  gl_Position = inPosition;
  myColour.rg = inPosition.xy;
  myColour.b = 1.0;
}
```

What does this vertex shader do?

# Draw a triangle: fragment shader

- Recall: *fragment shader* is run once for every pixel on the screen that is covered by the graphic primitive.
- The fragment shader is run with inputs that (optionally) have already been interpolated between the vertices.
- Fragment shader should give (at least) an RGBA colour as output that is drawn on the screen.

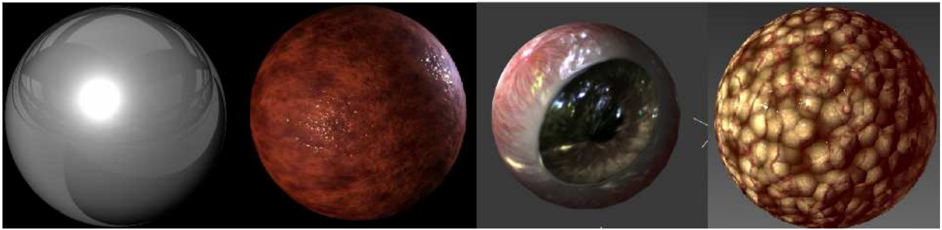
## Shader code

```
in vec3 myColour; // Remember myColour from last slide?
out vec4 pixel;

void main() {
  pixel.rgb = myColour;
  pixel.a = 0.0;
}
```

What does this fragment shader do?

# How flexible is it?

- Same CPU code for generating a sphere used with different vertex/fragment shaders.



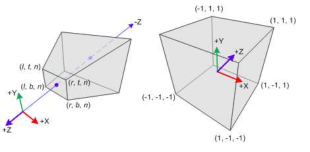
# Summary

- Colour perception, tristimulus model
- Basic rasterisation
- Programmable graphics pipeline
- Pixels and fragments
- Vertex and fragment shaders

# What's next

## Lecture 2: Essential algebra for graphics

- Tomorrow, 15.15–17.00
- T-211
- Hughes et al.: Chapters 7, 10, 11, 13




## Lab 1: OpenGL intro

- Wed 2 Nov, 08.15–12.00
- T-006

# References

Rafael C. Gonzales and Richard E. Woods (2007). *Digital Image Processing*. Third edition. Pearson.

Viktor Hallengren and Måns Granath (2016). *Virtual Classroom in Virtual Reality*. B.Sc. thesis, Örebro University.