Welcome to IN3005/INM376 Computer Graphics

Visiting lecturer: Eddie Edwards

What are computer graphics?

⇨ The digital synthesis and manipulation of visual data by a computer, often using specialised software and/or hardware

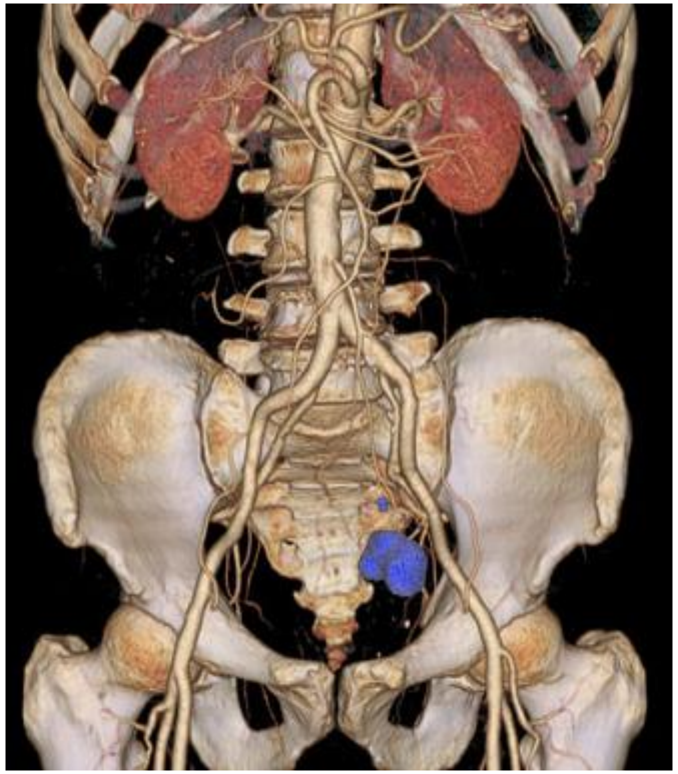
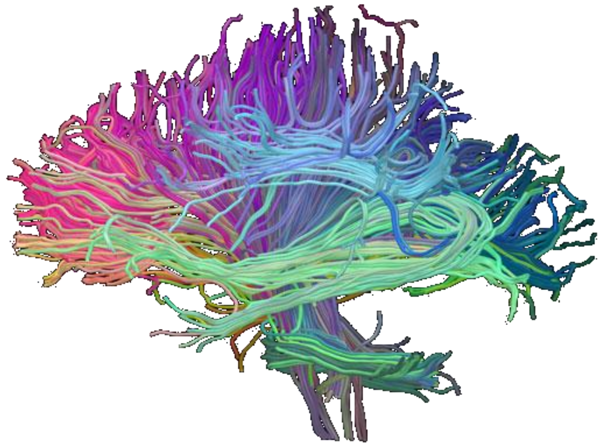
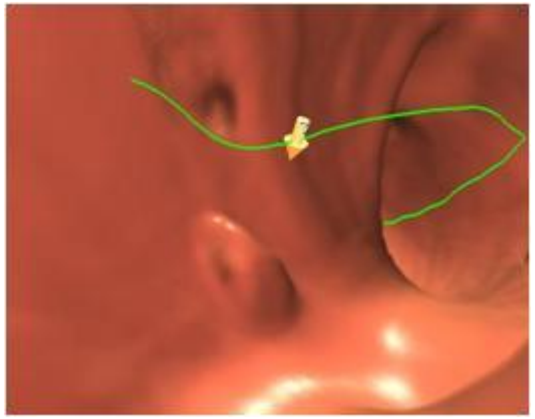
Computer graphics impact

The development of computer graphics has made computers easier to interact with, and better for understanding and interpreting many types of data.

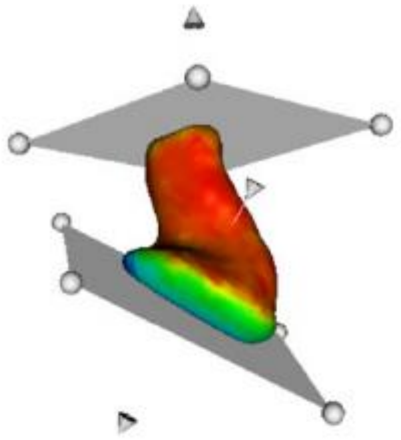
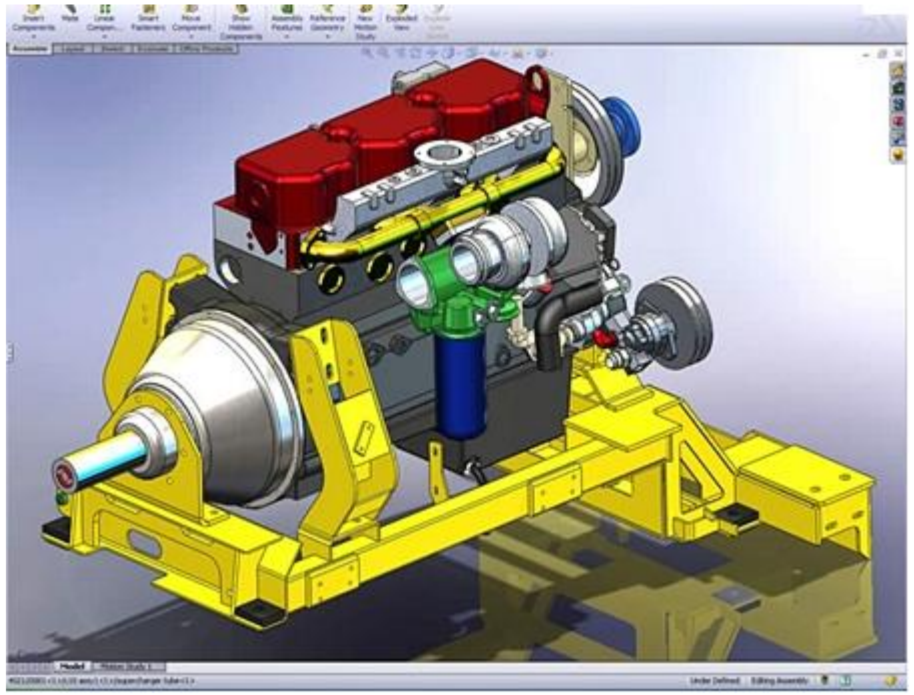
Developments in computer graphics have had a profound impact on many types of media and have revolutionised movies, medical imaging, product design, and the computer game industry.

Movies

Medical imaging

Design

Computer games

What is OpenGL?

⇨ Open Graphics Library (OpenGL) is cross-platform application programming interface (API) for developing applications that use 2D and 3D computer graphics. It provides a *software interface to graphics hardware*. It is **not** open source software, merely a open specification.

- Wide availability (ish)
  - OpenGL: PC / workstation support: Windows, Linux, Mac OS (deprecated)
  - OpenGL ES for mobile computing: Android, Mac iOS (deprecated)
  - WebGL for 3D graphics in a compatible browser: Firefox, Chrome, Safari, Opera, Edge

What you will learn in this module

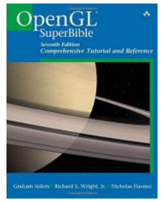
Fundamentals of computer graphics

- Mathematics of 3D graphics
- Geometric primitives and modelling
- Vertex buffer objects / vertex array objects
- Synthetic camera model
- GLSL, the OpenGL Shading Language
- Shader (GPU) programming
- Lighting, colour, materials, shading, shadows
- Texture mapping including multi-texturing
- Animation techniques
- Advanced topics

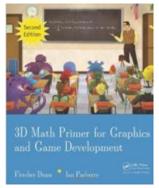
⇨ All of this in the context of OpenGL; coding in C++ and GLSL

References

There are numerous books on OpenGL and computer graphics. Here are a few recommendations, all of which are in the library

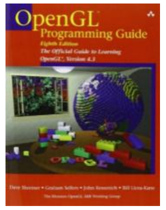


OpenGL Superbible, (Seventh Edition) Sellers et al.



3D Math Primer for Graphics and Game Development, (Second Edition) Dunn and Parberry. Note: Free online access through library!

OpenGL 4 Shading Language Cookbook, (Second Edition) Wolff. Note: Free online access through library!



OpenGL Programming Guide: The Official Guide to Learning OpenGL, Shreiner et al.

Administrative details

Lectures
- Tuesday, 6pm – 7.50pm : ELG06 (MAY MOVE SOON?)

Labs
- UG&PG students: Tuesday, 8 – 8.50pm : ELG06 (MAY ALSO MOVE)
-

Moodle4: IN3005 and INM376 Computer Graphics Supermodule
- Module website
- Lecture slides, sample code, handouts, resources

No office hours as I am not on site

I am generally available on email and can organise video chats on request

Email: Philip.Edwards@city.ac.uk

Administrative details

Please try to be engaged during lectures and labs

Take part in voting exercises – all of these are anonymous

Lectures and labs should be interactive

Please don't hesitate to ask questions (chat or raise hand)

My expectations of you:

- Attend **all lectures** and **all lab** sessions
- Lecture attendance is monitored
- Labs are vital to your progress on the later coursework
- Prepare by reading handouts and doing worksheets *in advance*
- Arrive on time
- Participate
  - Verbally, discussing/asking questions, taking notes
  - Working out examples
- Bring a pen / pencil and paper to lecture (or take notes on your laptop)

Assessment

Marks will be based on

The course is 100% coursework assessed

- Interim coursework – due end of reading week, Sunday 8th March - 25 %
- Demo and presentations of coursework
    - Video/Code submission – Sunday 12th April
    - Live Demos - Tuesday 11th April – formative
        - Replaces all teaching 7th April
        - *Attendance is compulsory!*
- Final report and code submission – Due Sunday 10th May – 75%

UG and PG students will have separate assessments (but you are welcome to attend the demo sessions of each other)

Coding

Visual Studio 2019/2022 (VS 2019/2022)

- Visual Studio Community is available for free, and it is a full-featured IDE: https://www.visualstudio.com/downloads/

- Nshader is useful for GLSL syntax highlighting https://marketplace.visualstudio.com/items?itemName=DanielScherzer.GLSL

Code samples will be available on Moodle as term progresses.

Study and use of sample code from the internet is *encouraged* but **must be referenced** if you include it in your work.

Coding

From the module description, you should be spending ***10-12 hours each week*** outside of lecture/lab studying and implementing. Find sources of code and perform your own implementations to expand on things we learn in the module.

Good places to look for modern (OpenGL 4) tutorials:

- http://www.swiftless.com/opengltuts/opengl4tuts.html
- http://www.mbsoftworks.sk/index.php?page=tutorials&series=1
- http://ogldev.atspace.co.uk/
- http://opengl.datenwolf.net/gltut/html/index.html (Learning Modern 3D Graphics Programming (Jason L. McKesson))
- http://www.opengl-tutorial.org/

- Don't forget the specification(s) and documentation!
- http://www.opengl.org/sdk/docs/man4/
- https://www.opengl.org/sdk/docs/man4/index.php
- http://glm.g-truc.net/0.9.6/glm-0.9.6.pdf

Overview of today’s lecture

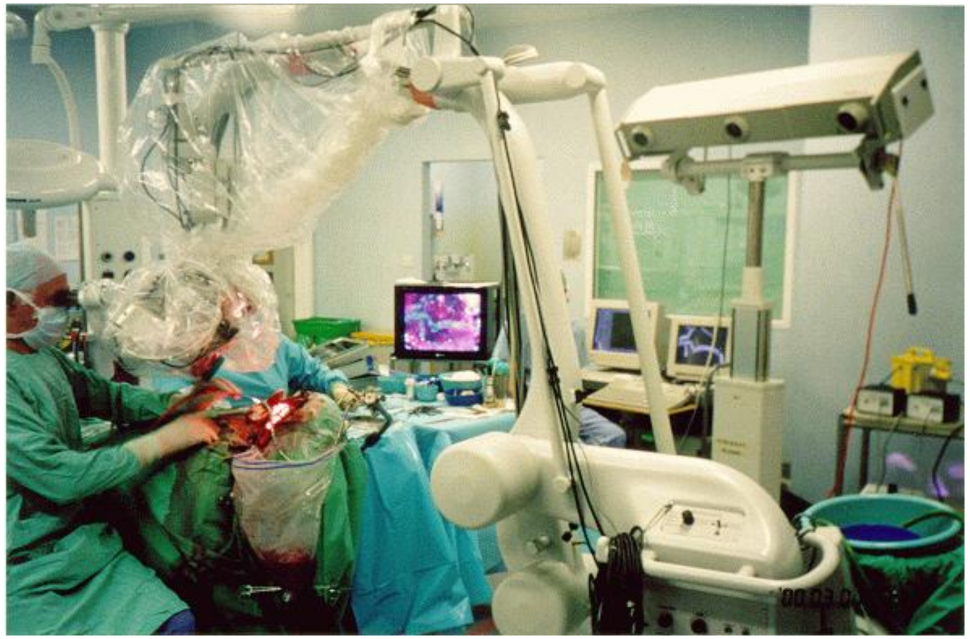
Introductions
Approaches to computer graphics
OpenGL programmable graphics pipeline
Vectors
A first (modern) OpenGL program in Windows

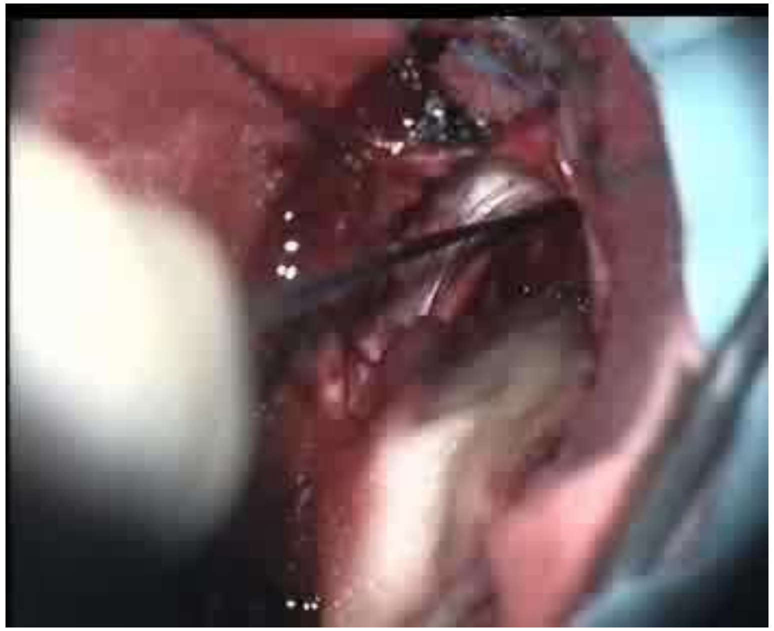
Introductions

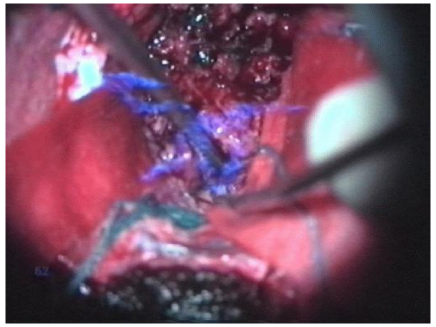
About me:

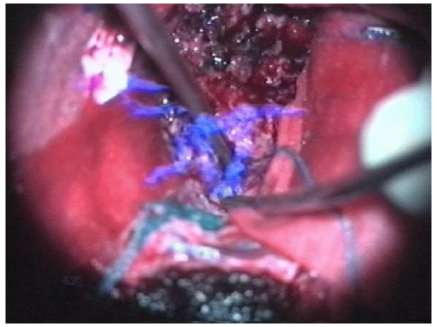
- PhD in image-guided intervention, KCL 2002
- 10 years as researcher at KCL
- Lecturer at Imperial College, 2004-2013
- Research themes:
  - Preoperative image segmentation and model building
  - Registration (image-image and image to physical space)
  - Surgical navigation
  - Augmented reality guidance
- Image analysis consultant 2013-2017
- Senior research associate, UCL, robotics and imaging, 2018-2022
- Associate Professor (Teaching), UCL, ML and robotics, 2022-…

assisted guided intervention

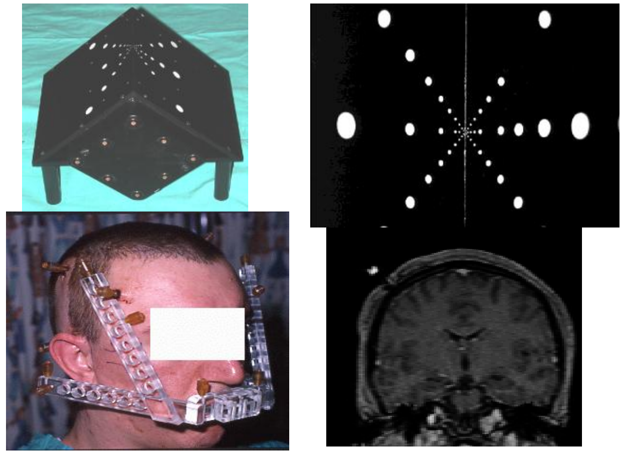




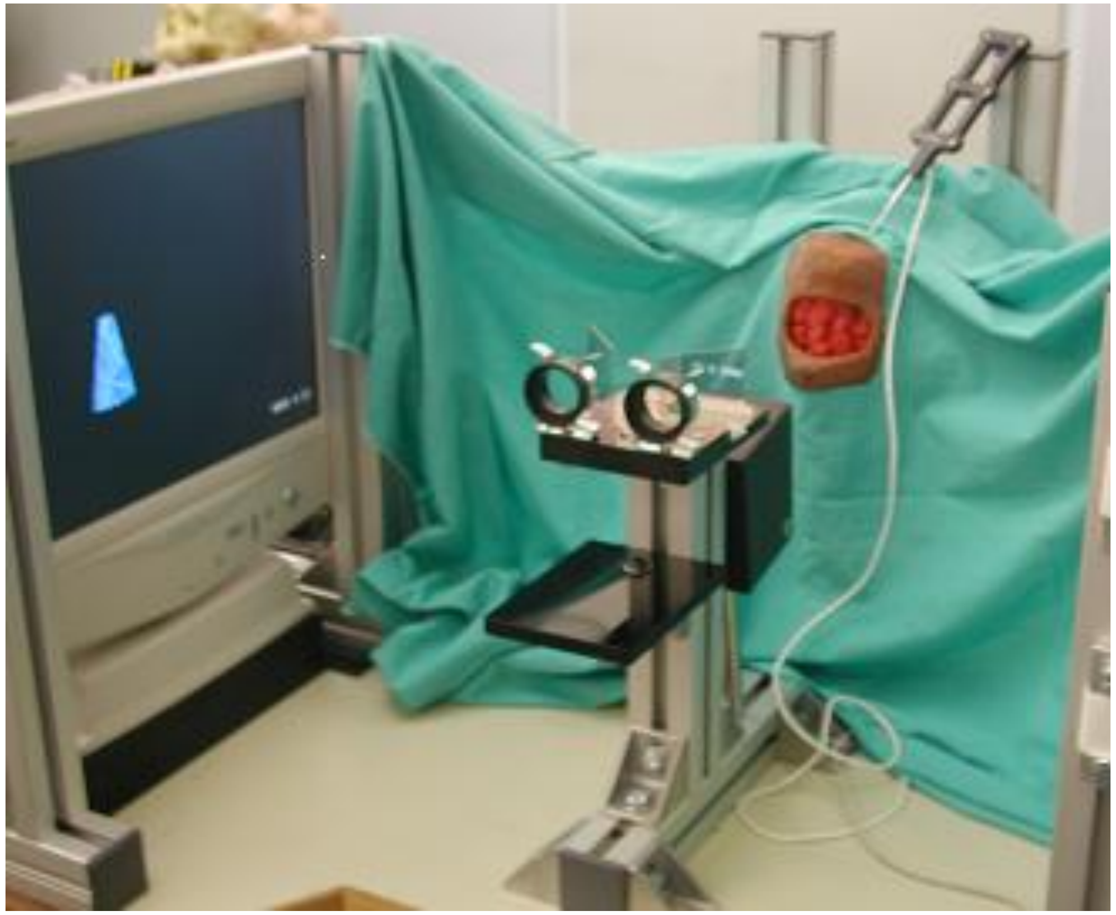


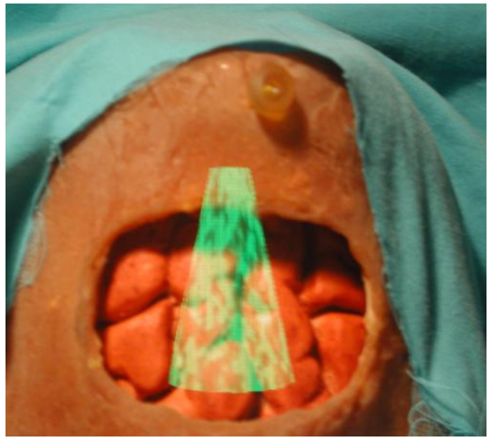


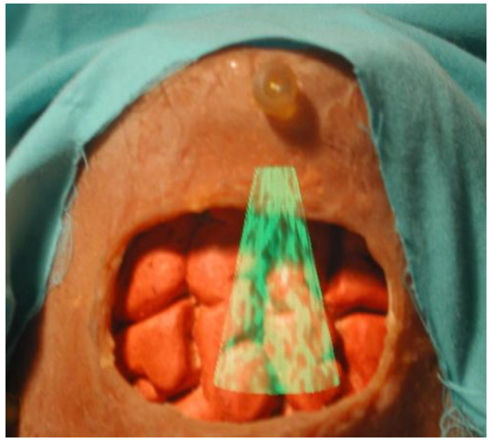
Edwards et al, IEEE TMI 2000



perception in stereo augmented reality

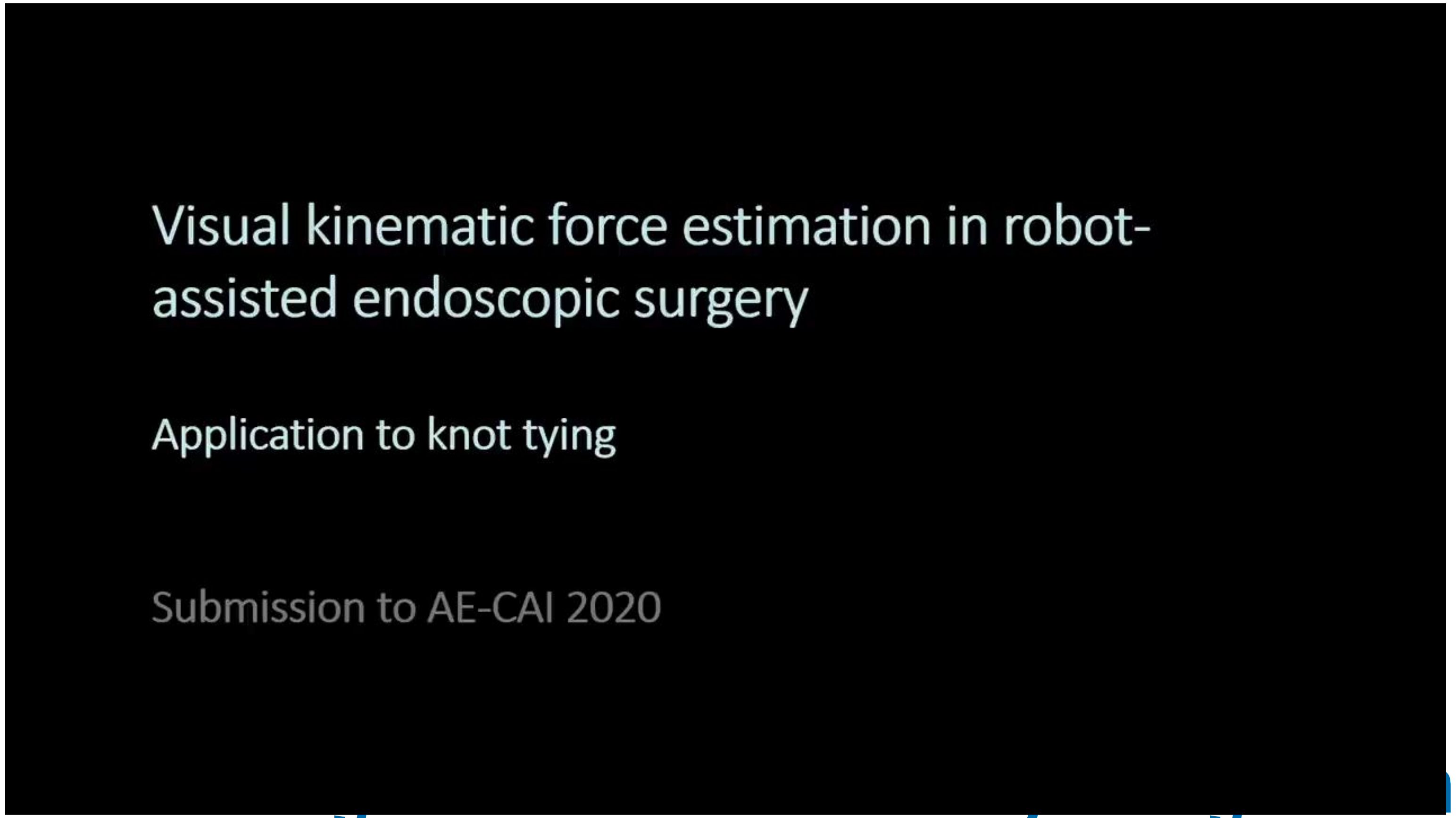






Laura Johnson et al, MMVR 2003

Current work in surgical robot vision



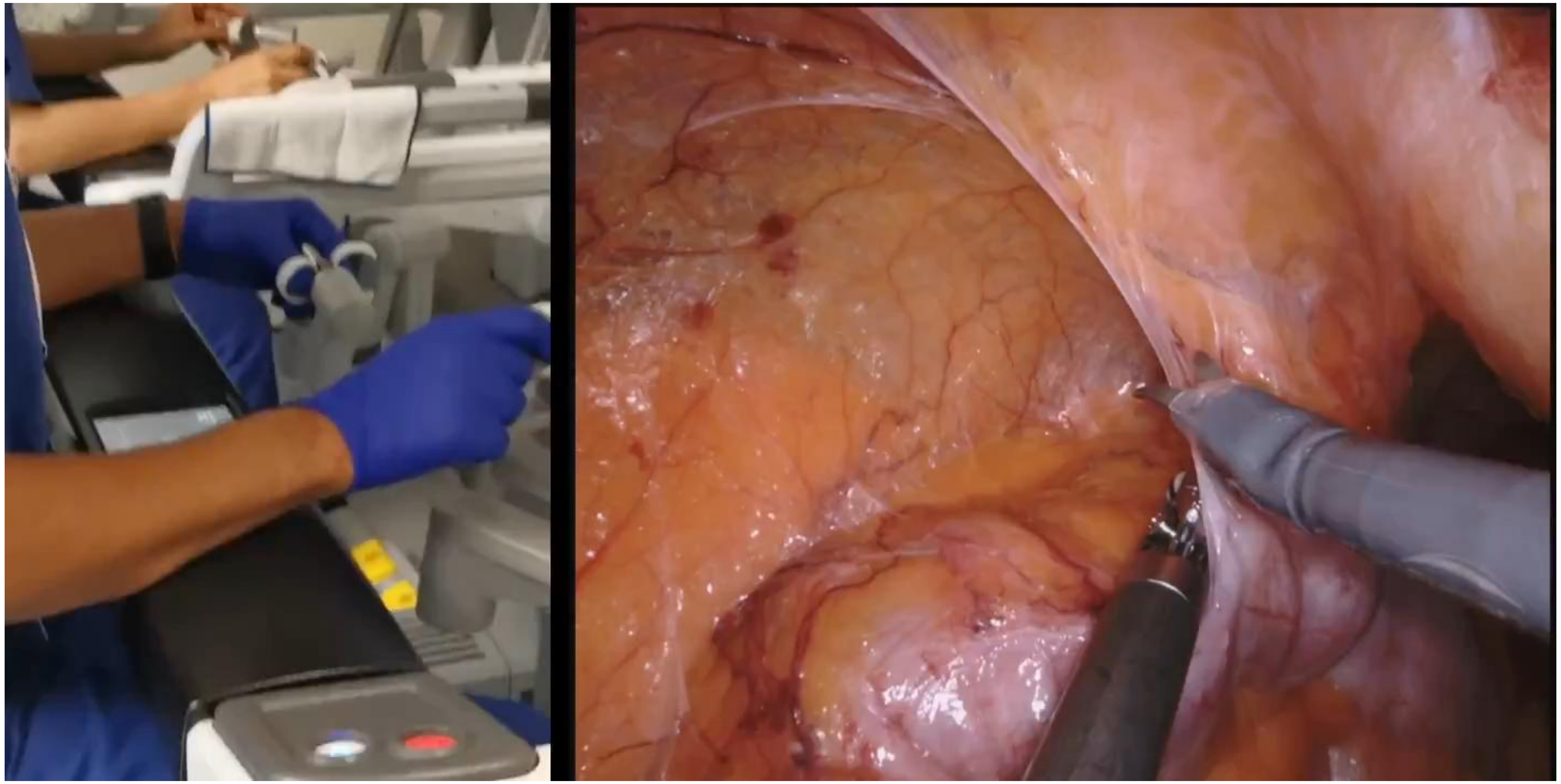
Visual kinematic force estimation in robot-assisted endoscopic surgery

Application to knot tying

Submission to AE-CAI 2020

and kinematics

Hand tracking in surgical robotics

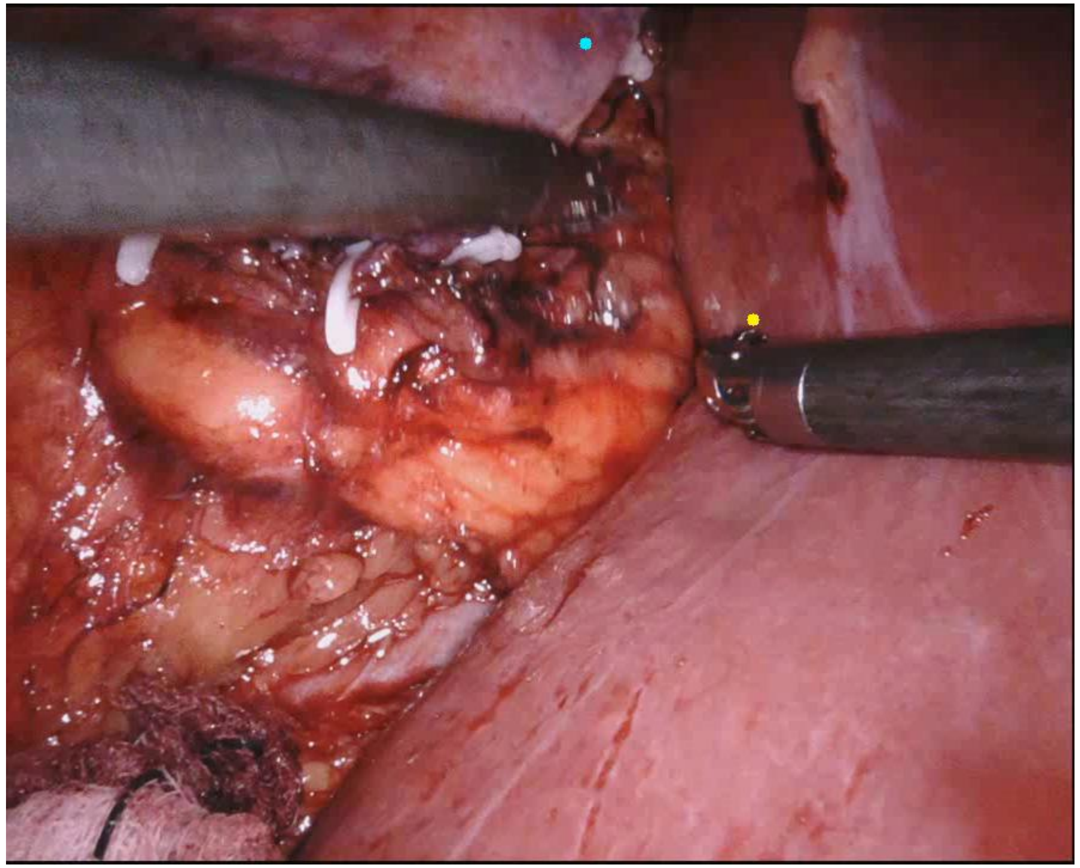


Projects Areas Offered

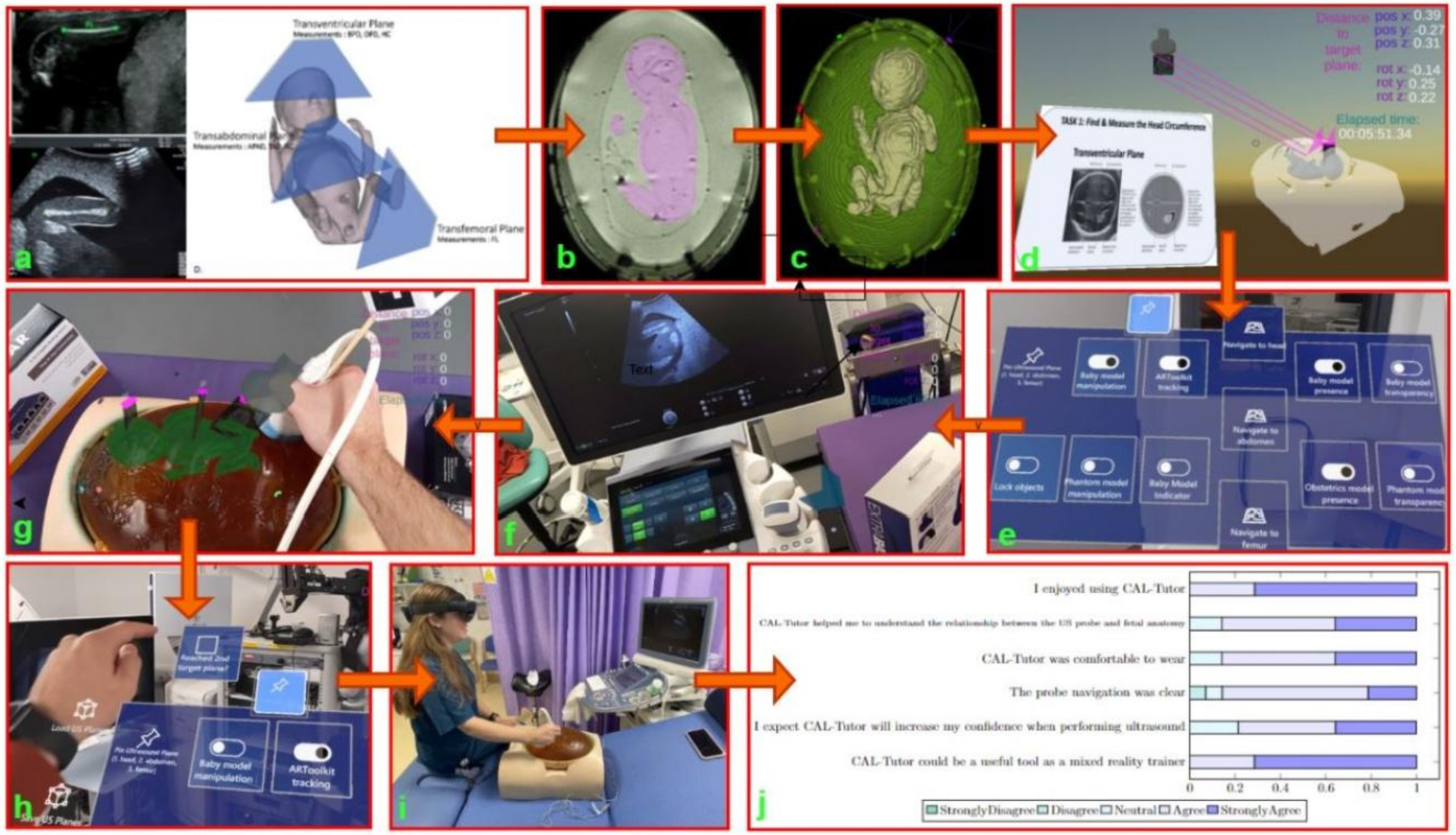
- Computer Graphics
- Virtual/Augmented Reality (Vive/HoloLens)
- ML applied to Computer Vision (Calibration, Tracking, Robotics and more)
- General Computer Games
- Any of the above applied to medical applications

Kinematic and visual tool tracking in robotic-assisted laparoscopic partial nephrectomy (RALPN)

- Recordings available with kinematics from real kidney surgery procedures
- Accurate hand-eye from calibrated stereo
- Tool segmentation - ideally from deep learning
- Integration of kinematics and stereo video for tool tracking
- Rolling hand-eye could give force estimation



HoloLens Guidance for Ultrasound Training


a
Transventricular Plane
Transabdominal Plane
Transfemoral Plane
b
c
d
TASK 1: Find & Measure the Head Circumference
Transventricular Plane
Distance to target plane:
pos x: 0.39
pos y: -0.27
pos z: 0.31
rot x: -0.14
rot y: 0.25
rot z: 0.22
Elapsed time: 00:05:51.34
e
Navigate to head
Baby model manipulation
ARToolkit tracking
Baby model presence
Baby model transparency
Navigate to abdomen
Lock objects
Phantom model manipulation
Baby Model Indicator
Obstetrics model presence
Phantom model transparency
Navigate to femur
f
g
h
Reached 2nd target plane!
Baby model manipulation
ARToolkit tracking
Load US Planes
Save US Planes
i
j
I enjoyed using CAL-Tutor
CAL-Tutor helped me to understand the relationship between the US probe and fetal anatomy
CAL-Tutor was comfortable to wear
The probe navigation was clear
I expect CAL-Tutor will increase my confidence when performing ultrasound
CAL-Tutor could be a useful tool as a mixed reality trainer
0
0.2
0.4
0.6
0.8
1
StronglyDisagree
Disagree
Neutral
Agree
StronglyAgree

Summary of projects

- All involve 3D graphics, many with OpenGL
- 3D graphics from medical images (MRI/CT)
- Alignment of models to the surgical scene (registration)
- Augmented reality (AR) visualisations
- Investigation of AR perception
- Combination of computer vision and computer graphics

- If any PG students may be interested in an individual project in graphics – particularly related to AR or medical applications – please get in touch

3D graphics

How it is done:

- Modelling: Geometry, lighting, materials, textures
- Animation: Moving the camera / objects in the scene
- Rendering: Synthesizing the image

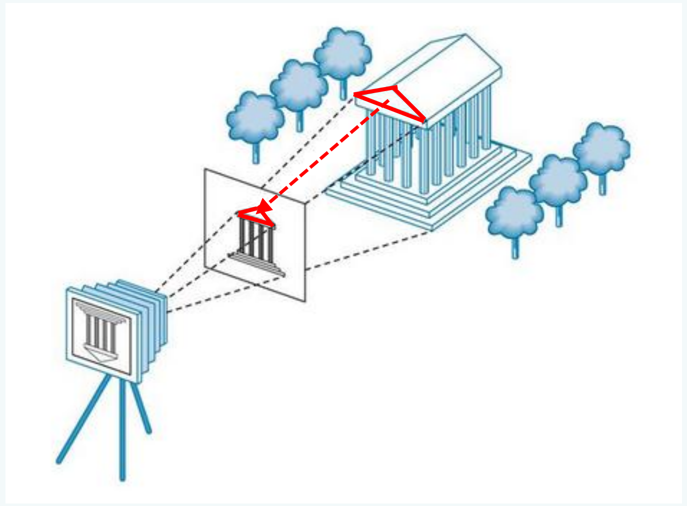
GRAN TURISMO
THE REAL DRIVING SIMULATOR

Gran Turismo Sport

Standard approaches to rendering

Rasterisation (object order)

Projects each 3D object from scene to image plane
3D polygons project to 2D polygons
2D polygons filled in (rasterised)

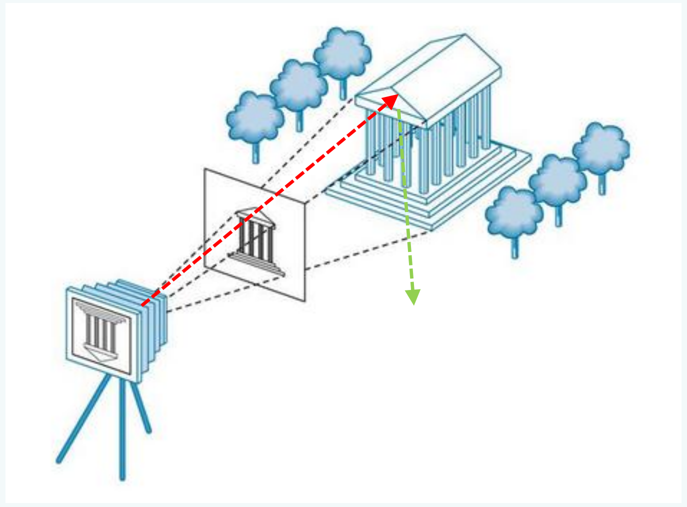


Typically used in computer games, as this method is very fast, supported by GPUs

Standard approaches to rendering

Ray tracing (pixel order)

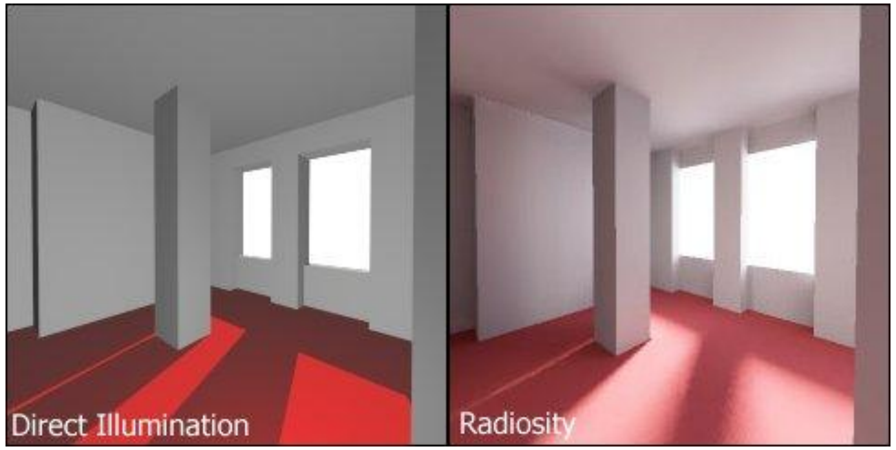
Back-projects rays through each pixel into scene
Determine where ray hits 3D objects in scene
Multiple bounces possible



Simulates the flow of light, allowing for more realistic rendering of complex effects. Typically used in movies.

Standard approaches to rendering

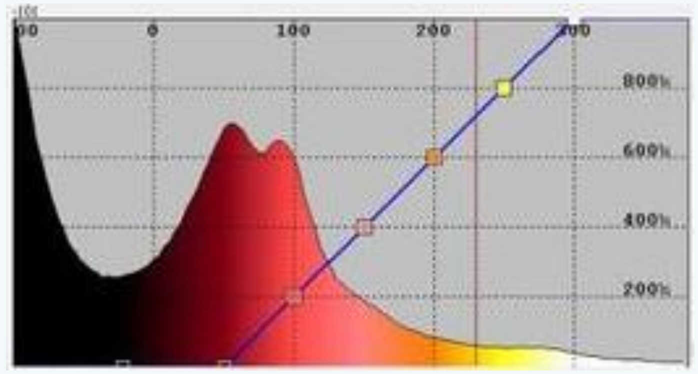
Radiosity (global illumination)
Computes the amount of light energy transferred among surfaces in the scene.
Divides surfaces to be rendered into patches Computes a "form factor" describing how much light is transferred between patches
Simulates how light is diffusely reflected from different objects in the scene. For example, the image on the right, the walls have a slightly reddish hue from diffuse reflection from the floor



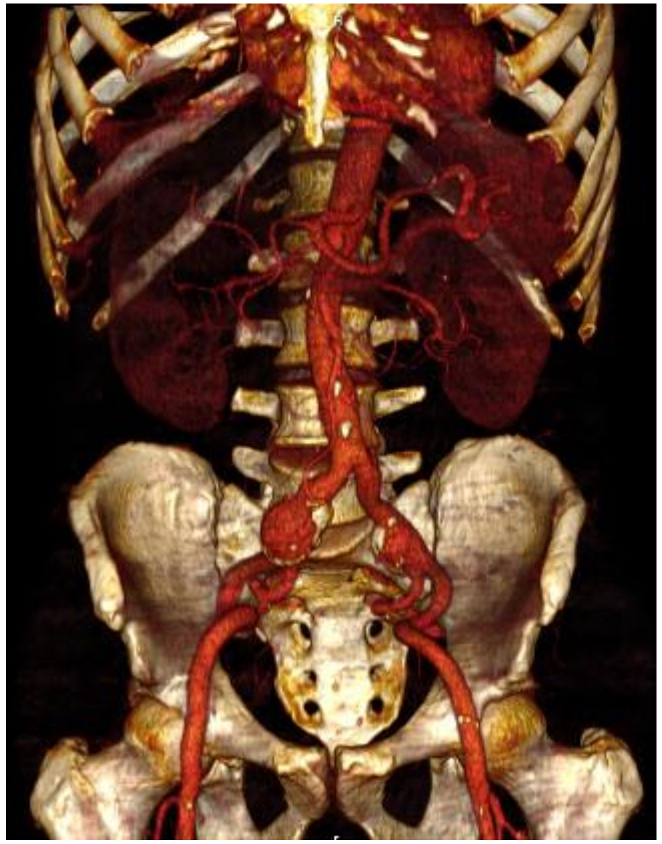
Standard approaches to rendering

Volume rendering

Shoots rays from the camera into the scene, which is composed a many voxels (3D cubes). Each voxel has a colour and transparency value; these are composited along the ray.



This is a common approach in medical imaging, but has much wider applications.



VRRenderWebSurg

OpenGL history

- Originally developed at Silicon Graphics (SGI) as a multi-purpose, platform independent graphics API
- From 1992: Development overseen by architecture review board (ARB)
- From 2006: Control passed to the Khronos group
- Versions:
  - 1.x, 2.x, 3.x, 4.x (latest version 4.6)
  - Shaders were introduced in OpenGL 2.0. Shaders are programs that run on the GPU. They are programmed in a separate language (GLSL) that is like C.
  - Evolving standard and architecture; deprecation introduced in OpenGL 3.0.
  - Advanced features available through extensions

OpenGL vs Direct3D/Metal

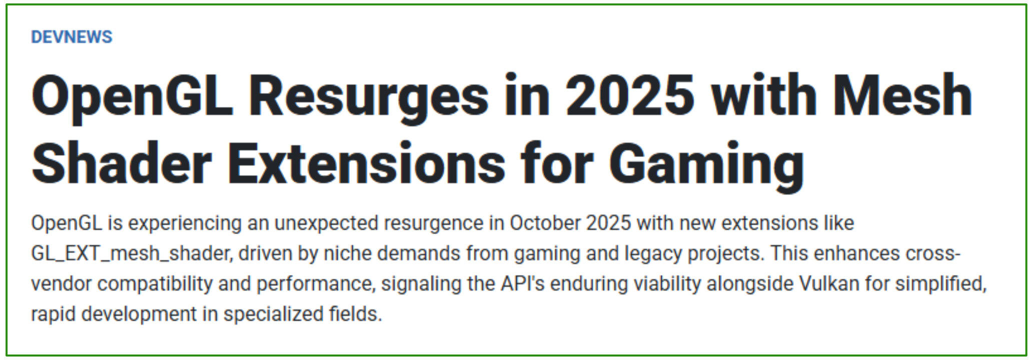
- Direct3D (part of DirectX) is another API for 3D graphics
- Direct3D is
  - A proprietary standard owned by Microsoft
  - Used extensively on Windows and Xbox for gaming
- Metal is a similar API from Apple
- On non-Windows systems, OpenGL **was** the defacto standard
  (deprecated by Apple)
- OpenGL is
  - Cross-platform (for now)
  - Portable (for now)
  - Open standard
  - Allows hardware manufacturers to compete through extensions

Vulkan

- Vulkan is a newer API for 3D graphics, which was released about a year ago.
- The API is also a standard like OpenGL, and maintained by the Khronos group.
- Compared to OpenGL, Vulkan is lower overhead and gives the programmer more direct control of the GPU. It is also designed to reduce CPU workloads. It's been described as "close to the metal". Typically this means an increase in *performance* but also *coding complexity*.
- Vulkan is *not* the next version of OpenGL, and it does not replace OpenGL. In fact, Vulkan will support GLSL, so expect Vulkan and OpenGL to coexist.
- http://www.toptal.com/api-developers/a-brief-overview-of-vulkan-api

A few well-known games using OpenGL

- Minecraft
- Dota 2
- Angry Birds
- No Man’s Sky
- Portal series (Mac)
- Quake series
- Star Wars: Knights of the Old Republic
- Unreal Tournament



DEVNEWS

OpenGL Resurges in 2025 with Mesh Shader Extensions for Gaming

OpenGL is experiencing an unexpected resurgence in October 2025 with new extensions like GL_EXT_mesh_shader, driven by niche demands from gaming and legacy projects. This enhances cross-vendor compatibility and performance, signaling the API's enduring viability alongside Vulkan for simplified, rapid development in specialized fields.

Minecraft

No Man’s Sky

Dota 2

What OpenGL doesn’t do

- OpenGL only provides rendering functions. No concept of
  - Windowing systems
  - Audio
  - Printing
  - Keyboard / mouse
  - Input devices (joysticks, etc).

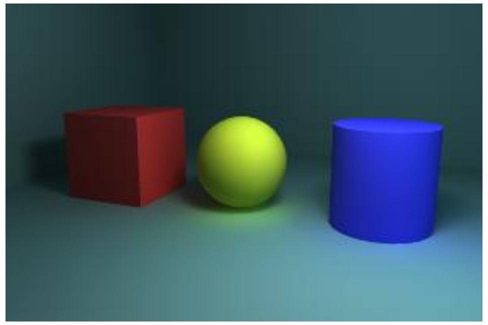
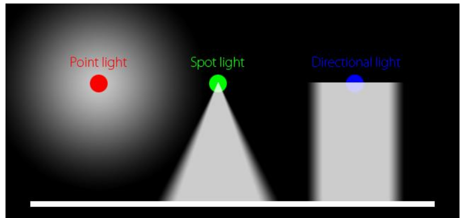
  ⇨ This way, OpenGL is independent of the OS
- OpenGL can be integrated directly (WGL) or through add-on APIs (GLUT, freeGLUT, GLFW, SDL, etc.). In this module, we’ll be using WGL to create a *rendering context*, which connects OpenGL to a window.

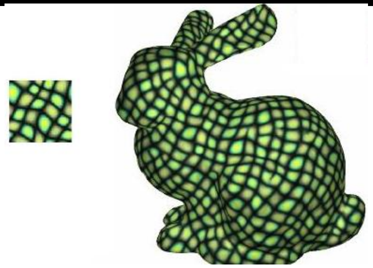
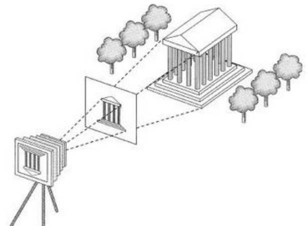
Deprecation

- Over the years, functionality kept being added to OpenGL, to the point where it became rather bloated.
- In 2008, OpenGL 3.0 was released. In this version, older OpenGL methods were deprecated, and only the newest and best (fastest, flexible) ways of rendering were kept.
- In 2009, OpenGL 3.1 was released. In this version, all deprecated functionality was dropped from the OpenGL *core*.
- However, there is **a lot** of legacy code in existence. Rather than make all these applications obsolete, OpenGL supports a *compatibility context* where deprecated functions are still available.
- Modern OpenGL programming does not use these older deprecated functions.

Some terminology

Scene	The collection of all objects in space and their organisation
Primitives	The geometric form of objects in the scene, represented as collections of rudimentary shapes (triangles, lines, points)
Lights	Illumination in the scene; e.g. ambient, spotlights, area lights
Material	The way a primitive reflects light, including its colour
Texture	An image applied to primitive
Camera	A mathematical model to transform 3D geometry into a 2D image

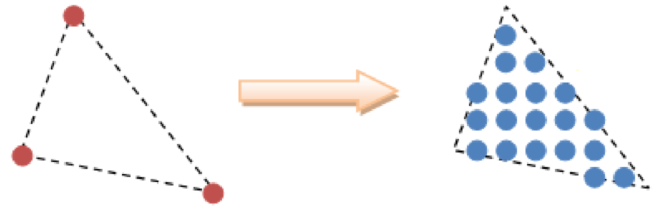



Rendering in OpenGL

- What is the OpenGL rendering pipeline?
  - A sequence of processes applied to raw data (meshes, vertices, etc.) to render a 3D image on a 2D screen

- How it works:
  - Project all 3D geometry onto the 2D image plane
    - Virtual camera model, geometric transformations
  - Determine the pixels to fill (fragments)
    - Primitive assembly, scan conversion
  - Compute color of each fragment
    - Li s, texture mapping



Each pixel to be filled is called a *fragment*

Fixed function vs. programmable pipeline

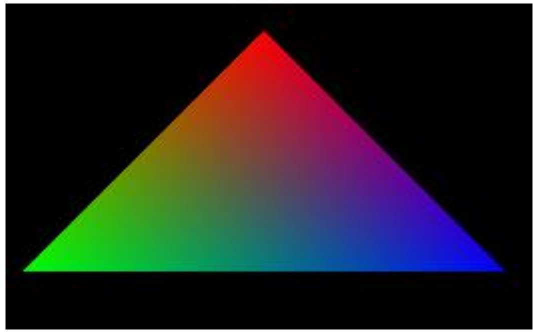
- OpenGL is an interface to the graphics hardware. It exposes features of the underlying graphics hardware to application developers.

- **Fixed function pipeline**
  - Graphics implemented through a set of pre-defined functions, such as transform and lighting (glRotatef, glLightfv, etc.)
  - Programmer is limited to what those functions can provide (e.g., 8 lights, Blinn-Phong model, etc.)

- **Programmable pipeline**
  - What you will learn in this module
  - Graphics hardware is programmable through *shader* programs, which are implemented in a shader language (GLSL)
  - Shader programs run on the GPU
  - This enables new possibilities for graphics – implementing things that are difficult or impossible to do using the fixed function pipeline

- OpenGL 3.0 was the last core version to support the fixed function pipeline, when it was deprecated. The fixed function pipeline was dropped in OpenGL 3.1 for the core context, but is still available through a compatibility context.

Legacy vs. modern OpenGL programming

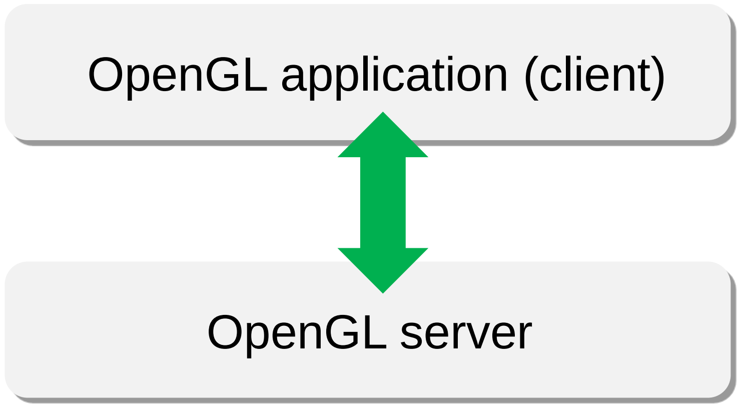


```
// Draw a triangle
glBegin (GL_TRIANGLES);
    glColor3f(1, 0, 0);
    glVertex3f(0, 1, 0);
    glColor3f(0, 1, 0);
    glVertex3f(-1, 0, 0);
    glColor3f(0, 0, 1);
    glVertex3f(1, 0, 0);
glEnd();
```

```
// Setup triangle vertex positions
fTriangle[0] = -1.0f; fTriangle[1] = 0.0f; fTriangle[2] = 0.0f;
fTriangle[3] = 1.0f; fTriangle[4] = 0.0f; fTriangle[5] = 0.0f;
fTriangle[6] = 0.0f; fTriangle[7] = 1.0f; fTriangle[8] = 0.0f;

// Setup triangle vertex colours
fTriangleColor[0] = 0.0f; fTriangleColor[1] = 1.0f; fTriangleColor[2] = 0.0f;
fTriangleColor[3] = 0.0f; fTriangleColor[4] = 0.0f; fTriangleColor[5] = 1.0f;
fTriangleColor[6] = 1.0f; fTriangleColor[7] = 0.0f; fTriangleColor[8] = 0.0f;

// Generate a VAO and two VBOs
glGenVertexArrays(1, uiVAO);
glGenBuffers(2, uiVBO);

// Create the VAO for the triangle
glBindVertexArray(uiVAO[0]);

// Create a VBO for the triangle vertices
glBindBuffer(GL_ARRAY_BUFFER, uiVBO[0]);
glBufferData(GL_ARRAY_BUFFER, 9*sizeof(float), fTriangle, GL_STATIC_DRAW);
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);

// Create a VBO for the triangle colours
glBindBuffer(GL_ARRAY_BUFFER, uiVBO[1]);
glBufferData(GL_ARRAY_BUFFER, 9*sizeof(float), fTriangleColor, GL_STATIC_DRAW);
glEnableVertexAttribArray(1);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0, 0);

// Load and compile shaders
shVertex.loadShader("data\\shaders\\shader.vert", GL_VERTEX_SHADER);
shFragment.loadShader("data\\shaders\\shader.frag", GL_FRAGMENT_SHADER);

// Create shader program and add shaders
spMain.createProgram();
spMain.addShaderToProgram(&shVertex);
spMain.addShaderToProgram(&shFragment);

// Link and use the program
spMain.linkProgram();
spMain.useProgram();
```

Client / server model

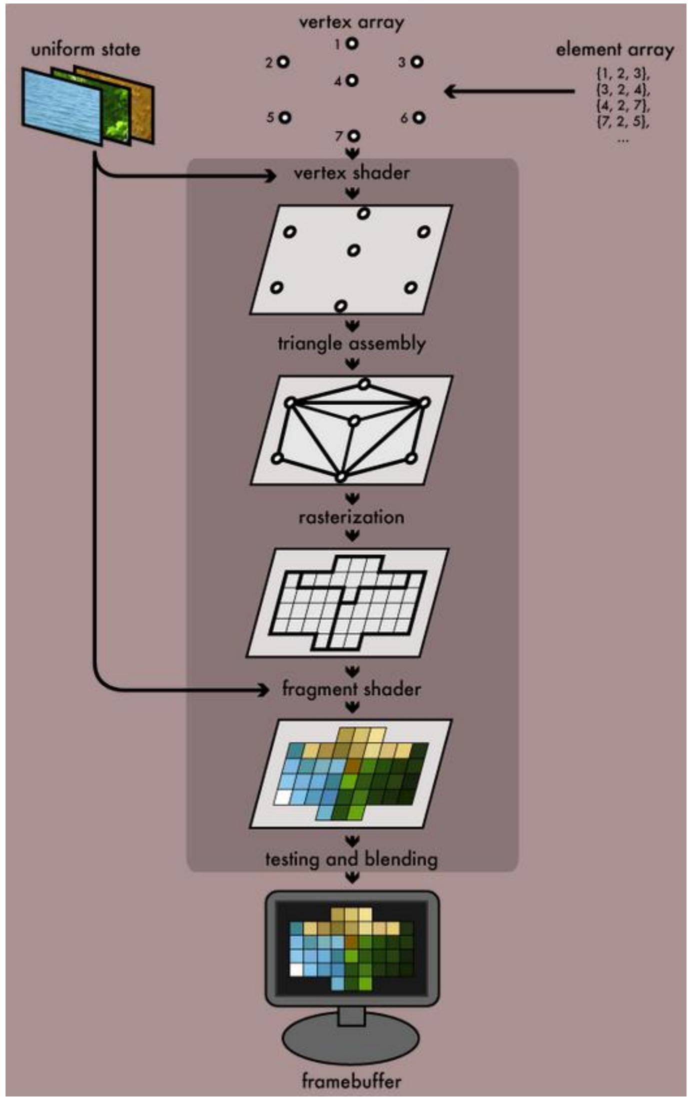
- OpenGL application (**client**) loads data (meshes, textures, shaders) from disk
- Client then transfers the data to the GPU (**server**)
- Shader programs run on the GPU



Runs on CPU, using RAM

Runs on GPU, using VRAM

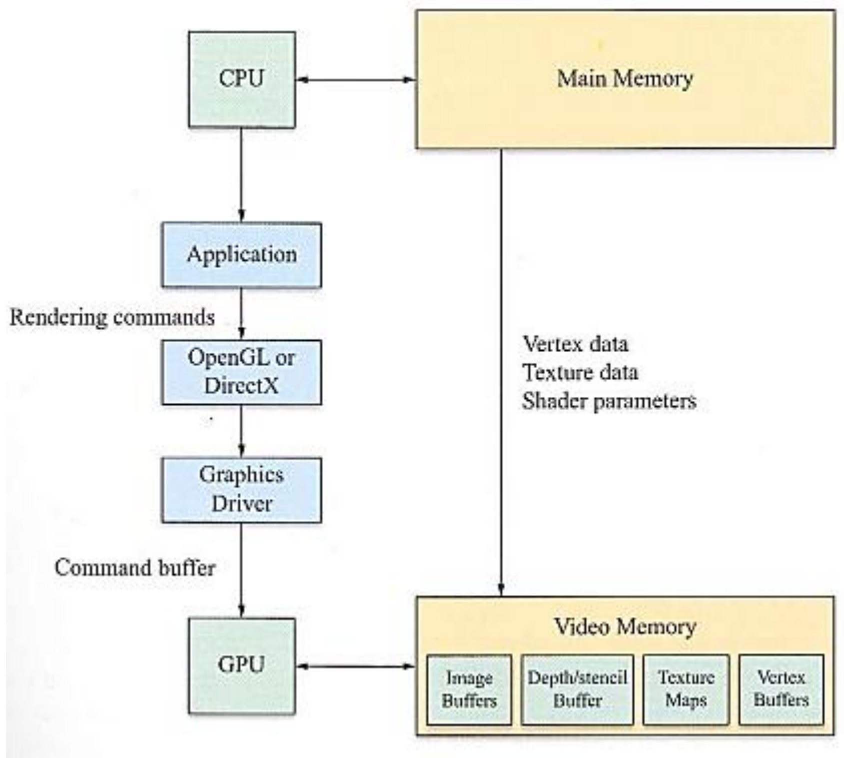
Modern rendering pipeline (simplified)



http://duriansoftware.com/joe/An-intro-to-modern-OpenGL.-Chapter-1:-The-Graphics-Pipeline.html

Rendering: CPU and GPU

- The graphics processing unit (GPU) is running asynchronously with the CPU.
- Running on the CPU, the application issues commands through OpenGL to the graphics driver, which in turn speaks to the GPU in its native language.
- On the graphics card there is video memory (VRAM) that stores data and images buffers where the synthesized image is formed.



[Lengyel, Ch1]

CITY
ST GEORGE'S
UNIVERSITY OF LONDON

Pipeline advantages and disadvantages

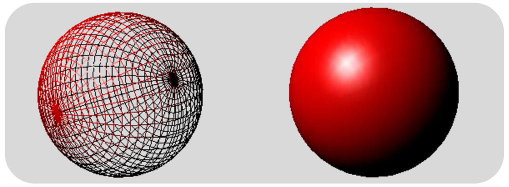
- The OpenGL pipeline is
  - Modular: vertices handled separately from fragments. This enables pipelining, where earlier stages can process data independently of later stages.
  - Local: Processing of primitives are done independently of each other. This allows parallelism of primitive rendering.

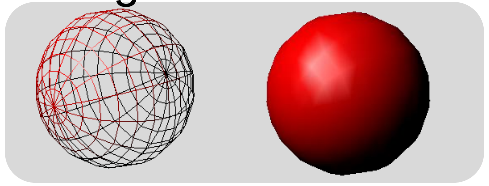
⇨ Only local knowledge of the scene is necessary to render using the OpenGL pipeline. However, non-local operations (like shadows, reflections, etc.) are not local and must be handled *through some other mechanism*.

Modern graphics programming

Modern graphics programming is designed to take advantage of graphics hardware. Shaders are a key component to this. In OpenGL 4.0, there are five types of shaders:

- **Vertex** shader: A program to process vertices (including rotation, translation, scaling, projection) and vertex attributes (lighting, colour, etc.). To render a triangle, the vertex shader is run three times (once for each vertex).
- **Fragment** shader: A program to fill in a fragment colour at a pixel in the rendered image. The fragment shader is run for every pixel in the primitive.
- **Geometry** shader: A program to create new from existing geometry. The geometry shader is optional, unlike the vertex and fragment shaders.
- **Two tessellation** shaders: Programs to change the tessellation of model.
  - Tesselation control shader
  - Tesselation evaluation shader


Higher tessellation


Lower tessellation

OpenGL 4.3 introduces

- **Compute** shader: A program for general purpose computing on the GPU.

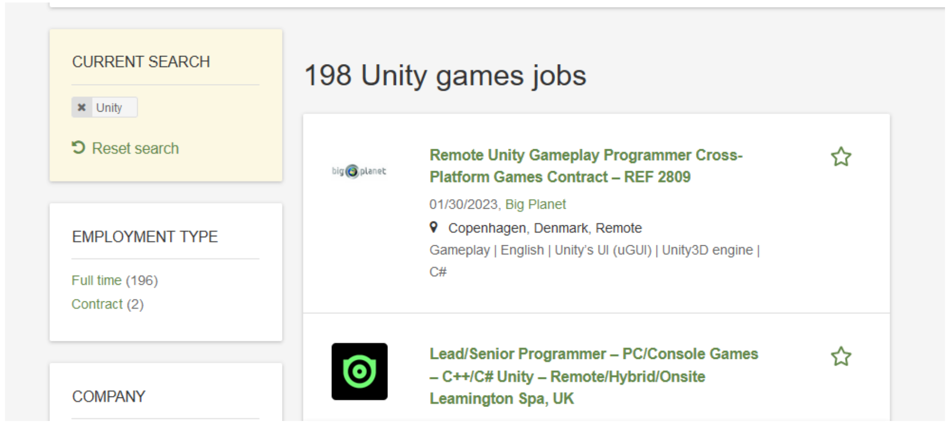
Mathematics

- Even with the support of existing source code, APIs, and GPUs, it is essential to have an understanding of the mathematics that underpin 3D computer graphics
- Mathematics fundamentals include
  - Vectors in two and three dimensions
  - Matrices
  - Trigonometry
  - Calculus

Mathematics at work in
Call of Duty: Vanguard

https://www.youtube.com/watch?v=OQ1CwPhE8KQ

Mathematics/C++ – game industry



CURRENT SEARCH

Unity

Reset search

EMPLOYMENT TYPE

Full time (196)

Contract (2)

COMPANY

198 Unity games jobs

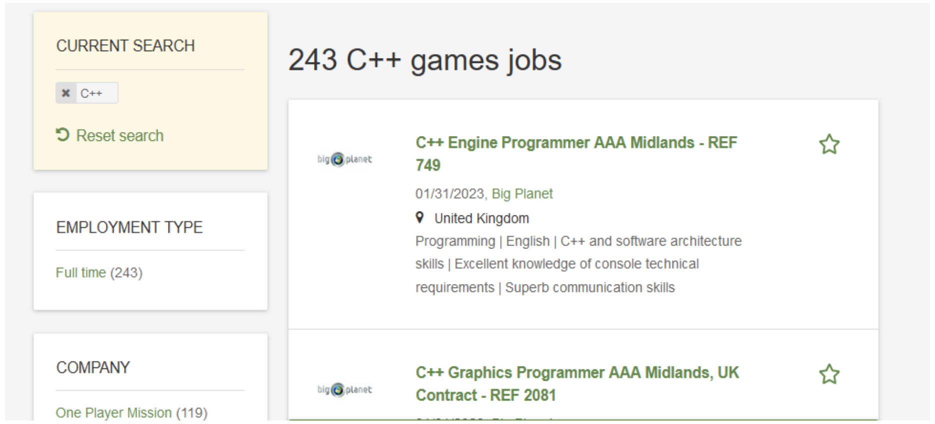
Remote Unity Gameplay Programmer Cross-Platform Games Contract – REF 2809

01/30/2023, Big Planet

Copenhagen, Denmark, Remote

Gameplay | English | Unity's UI (uGUI) | Unity3D engine | C#

Lead/Senior Programmer – PC/Console Games – C++/C# Unity – Remote/Hybrid/Onsite Leamington Spa, UK



CURRENT SEARCH

C++

Reset search

EMPLOYMENT TYPE

Full time (243)

COMPANY

One Player Mission (119)

243 C++ games jobs

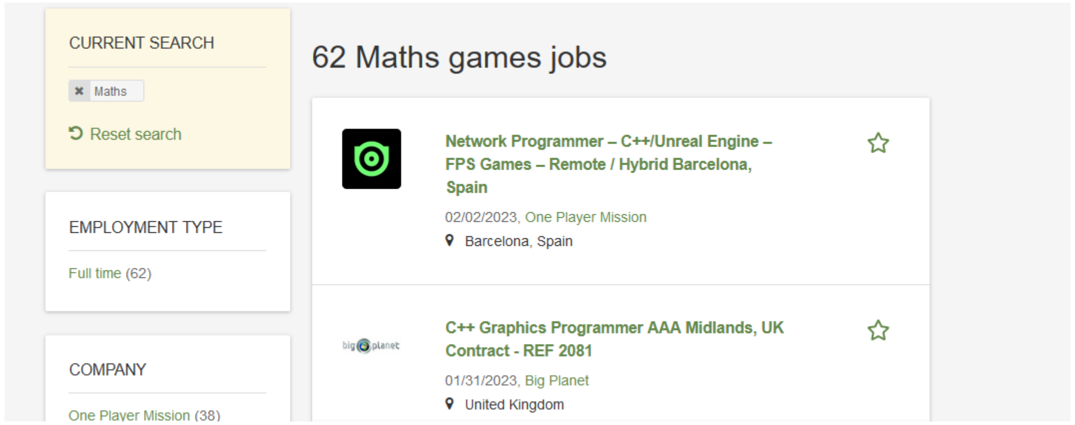
C++ Engine Programmer AAA Midlands - REF 749

01/31/2023, Big Planet

United Kingdom

Programming | English | C++ and software architecture skills | Excellent knowledge of console technical requirements | Superb communication skills

C++ Graphics Programmer AAA Midlands, UK Contract - REF 2081



CURRENT SEARCH

Maths

Reset search

EMPLOYMENT TYPE

Full time (62)

COMPANY

One Player Mission (38)

62 Maths games jobs

Network Programmer – C++/Unreal Engine – FPS Games – Remote / Hybrid Barcelona, Spain

02/02/2023, One Player Mission

Barcelona, Spain

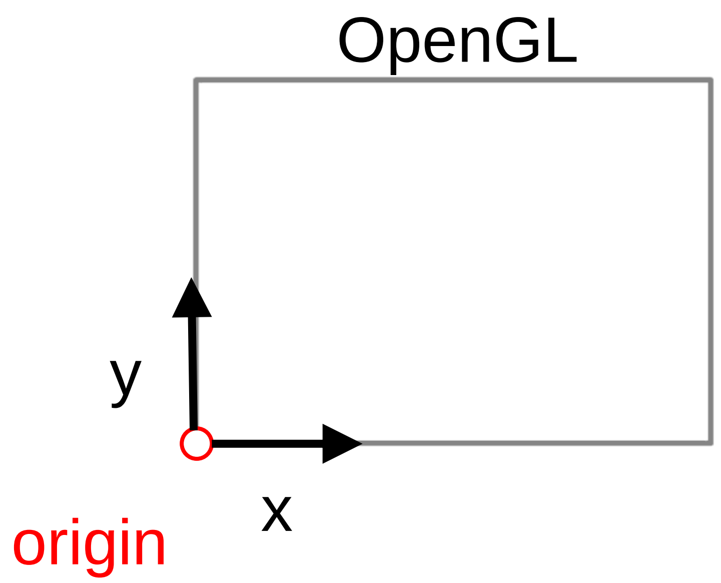
C++ Graphics Programmer AAA Midlands, UK Contract - REF 2081

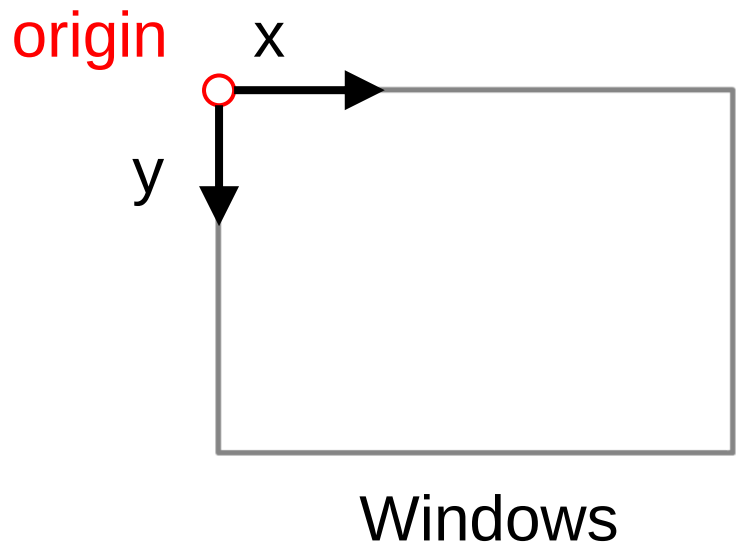
01/31/2023, Big Planet

United Kingdom

2D coordinates

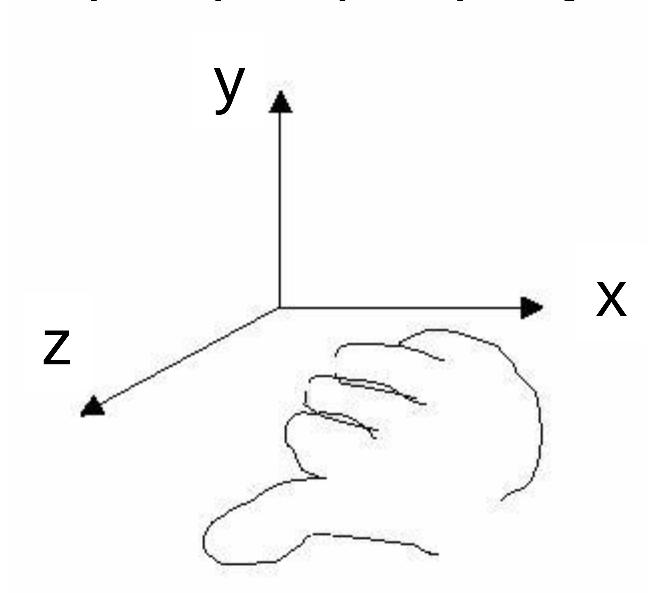
- Typically we use $[x, y]^T$ to represent a point in 2D space, such as the computer screen (display).
- The 2D coordinate system provides the origin and axes.
- Note: OpenGL display coordinates have
  - x-axis: right
  - y-axis: up (different from Windows!)





3D coordinates

- Typically we use $[x, y, z]^T$ to represent a point in 3D space.
- The 3D coordinate system provides the origin and axes.
- Typically one uses a right-handed coordinate system when programming OpenGL. This satisfies the *right hand rule*
  - This requires the *z* axis to be oriented in the direction your orientating your hand along the *x* fingers towards the *y* axis.



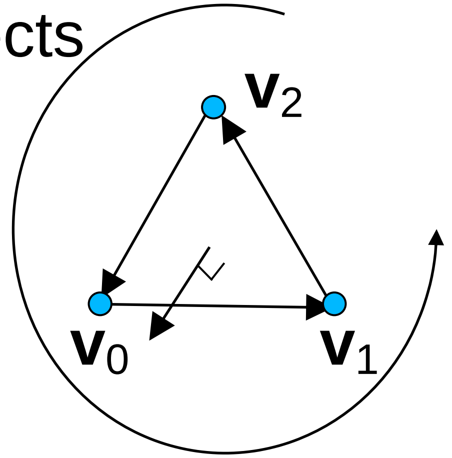
http://gra-phic.blogspot.co.uk/2011/07/3d-coordinate-systems-opengl.html

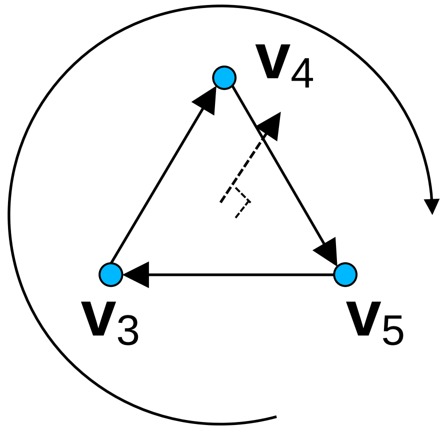
*the exception is the camera – more on this in a later lecture

Why the right hand rule is so ‘handy’

- The right hand rule is used to define orientation (winding) of planar objects

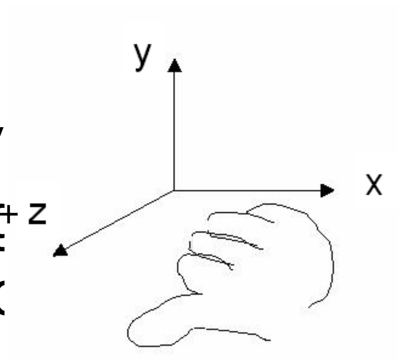
Counter-clockwise
(front facing)





Clockwise
(back facing)

- For rotations, if y ht thumb along the axis of rotation, your fing ound in the direction of a positive (anti-cloc ı.



Scalar

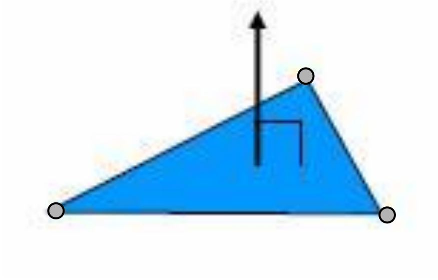
- A scalar is simply a number
- Mathematically, it is denoted with a non-bold font, for example
  - $x = 6.0$
  - $y = 5.0$
- Scalars can be added, subtracted, multiplied, and divided (except divide by 0):
  - $x + y = 11.0$
  - $x - y = 1.0$
  - $x * y = 30.0$
  - $x / y = 1.2$

Vector

- Notation: using a bold font
- Used to represent, in *N* (typically, 2, 3, or 4) dimensions:
  - Position of a point in space (from the origin)
  - Direction
  - Colour: E.g., red, green, blue
- An *N* dimensional vector can be written as

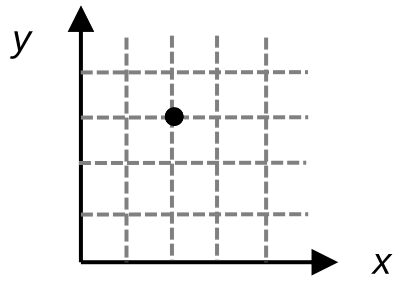
$$\mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_N \end{bmatrix}$$

components (or elements)



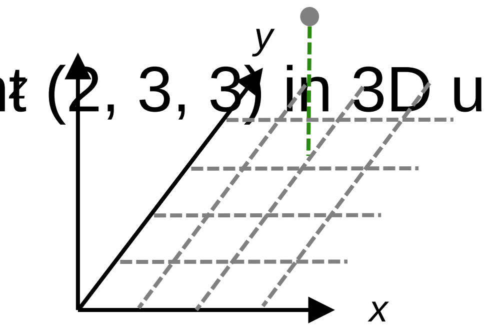
Vector examples

- Represent the point (2, 3) in 2D using a vector



$$\mathbf{p}_1 = \begin{bmatrix} 2 \\ 3 \end{bmatrix}$$

- Represent the point (2, 3, 3) in 3D using a



$$\mathbf{p}_2 = \begin{bmatrix} 2 \\ 3 \\ 3 \end{bmatrix}$$

Column vectors and row vectors

- A *column vector* is a vector that has all components in a vertical column:

$$\mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_N \end{bmatrix}$$

- A row vector has components in a horizontal row.

$$\mathbf{v}^T = [v_1, v_2, \cdots v_N]$$

- A *transpose* changes column vector to a row vector, and vice-versa.

Vector arithmetic

- Vectors can be added and subtracted, to form a new vector

$$\mathbf{p}+\mathbf{q}=\begin{bmatrix} p_1+q_1 \\ p_2+q_2 \\ \vdots \\ p_N+q_N \end{bmatrix} \qquad \mathbf{p}-\mathbf{q}=\begin{bmatrix} p_1-q_1 \\ p_2-q_2 \\ \vdots \\ p_N-q_N \end{bmatrix}$$

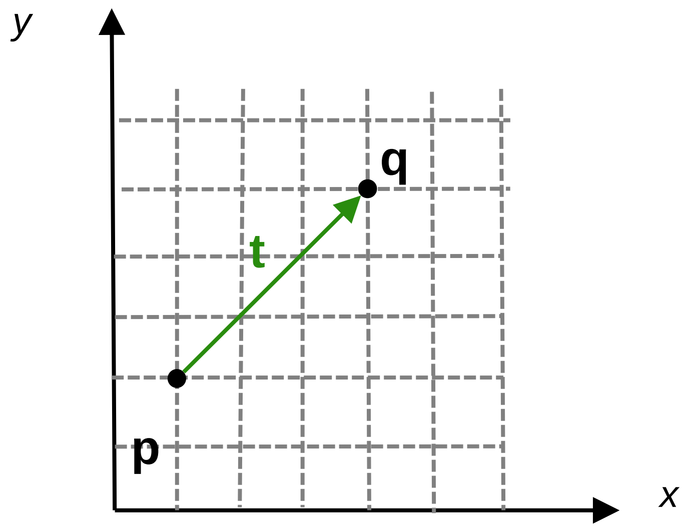
- For example, if **p** = [2, 2, 2]^T determine **p – q**.

$$\mathbf{p}-\mathbf{q}=\begin{bmatrix} 2 \\ 2 \\ 2 \end{bmatrix}-\begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}=$$

Vector addition and subtraction

- Often used for translation:
  - Example: move the point **p** = $[1, 2]^T$ by adding a displacement **t =** $[3, 3]^T$ to form the point **q**.

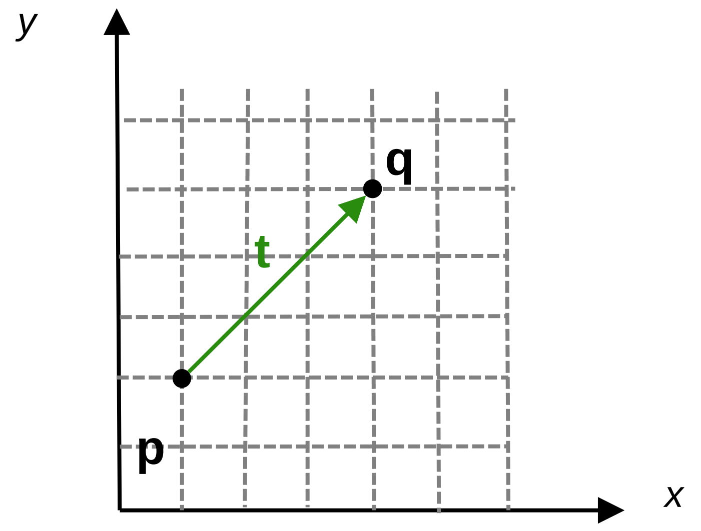
$$\mathbf{q} = \mathbf{p} + \mathbf{t} = \begin{bmatrix} 1 \\ 2 \end{bmatrix} + \begin{bmatrix} 3 \\ 3 \end{bmatrix} = \begin{bmatrix} 4 \\ 5 \end{bmatrix}$$



Vector addition and subtraction

- Also commonly used to find a vector between points:
  - Example: find the vector **t** between points **p** and **q** and originating from **p**.

$$\mathbf{t} = \mathbf{q} - \mathbf{p} = \begin{bmatrix} 4 \\ 5 \end{bmatrix} - \begin{bmatrix} 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 3 \\ 3 \end{bmatrix}$$



Scalar multiplication

- Scalar multiplication: multiplies each component of the vector by a scalar.

$$a\mathbf{v} = \mathbf{v}a = \begin{bmatrix} av_1 \\ av_2 \\ \vdots \\ av_N \end{bmatrix}$$

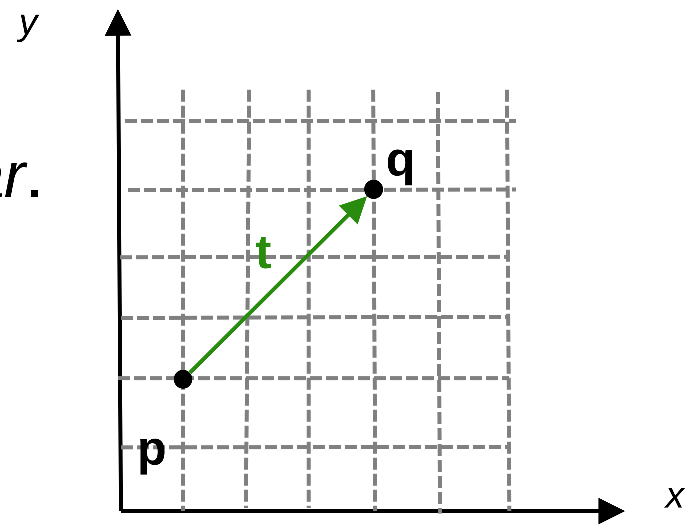
Vector length

- A vector has a length (or *magnitude*) given by the expression:

$$||\mathbf{v}|| = \sqrt{\sum_{i=1}^{N} v_i^2}$$

- Note that the vector length is a *scalar*.
- W $\mathbf{t} = \begin{bmatrix} 3 \\ 3 \end{bmatrix}$ length of **t?**

$$||\mathbf{t}|| = \sqrt{3^2 + 3^2} = \sqrt{18} = 3\sqrt{2}$$



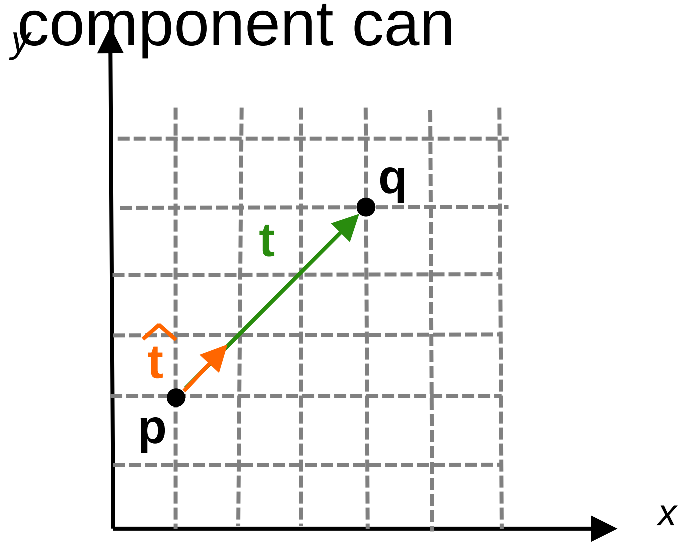
Vector normalisation

- Normalisation scales a vector so that it has unit length. That is, the length of the vector is one after normalisation.
- Any vector with at l $\frac{\mathbf{v}}{||\mathbf{v}||}$ one non-zero component can be normalised.

$\hat{\mathbf{t}}$

$$\hat{\mathbf{t}} = \frac{\mathbf{t}}{||\mathbf{t}||} = \frac{1}{3\sqrt{2}}\left[\begin{array}{c}3\\3\end{array}\right] = \left[\begin{array}{c}\frac{1}{\sqrt{2}}\\\frac{1}{\sqrt{2}}\end{array}\right]$$

- Determine , by normalising **t**



Exercise

- Find the vector **t**, starting at **p** = $[1, 1, 0]^T$ and ending at **q** = $[3, 3, 2]^T$
- What is the length of **t**?
- What is $\hat{\mathbf{t}}$, found by normalising **t**?

Hint: Draw it out on paper

$$\mathbf{t} = \mathbf{q} - \mathbf{p}$$

Useful equations:

$$||\mathbf{t}|| = \sqrt{\sum_{i=1}^{N} t_i^2}$$

$$\hat{\mathbf{t}} = \frac{\mathbf{t}}{||\mathbf{t}||}$$

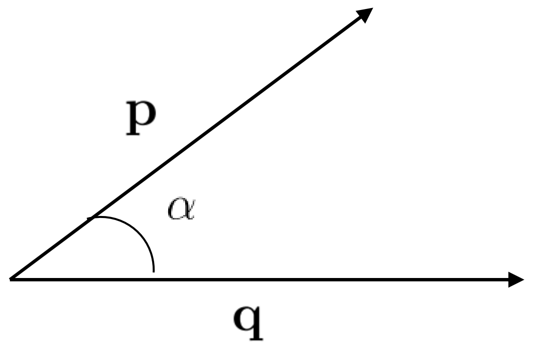
Dot product

- Dot product
  - Definition $\mathbf{p} \cdot \mathbf{q} = \sum_{i=1}^{N} p_i q_i$
  - Multiplies the *i*th component of each vector together, then takes the sum.
  - Note that the dot product is a *scalar*.
  - Example: What is the dot product of vectors **p** = $[3, 2, 1]^T$ and **q** = $[1, 0, -1]^T$?

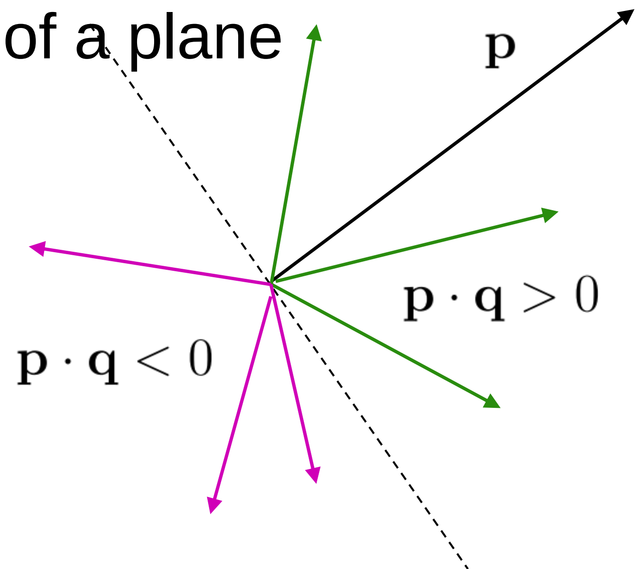
Answer: (3)(1) + (2)(0) + (1)(-1) = 2

Dot product

- Dot product is equivalent to $\mathbf{p} \cdot \mathbf{q} = ||\mathbf{p}|| \, ||\mathbf{q}|| \cos \alpha$
  -



◦ Special case: if $\mathbf{p} \cdot \mathbf{q} = 0$, the two vectors are *orthogonal*. This means they are perpendicular.
◦ The sign of the dot product tells us whether to the vectors lie on the same side or on opposite sides of a plane



$\mathbf{p}$
$\mathbf{p} \cdot \mathbf{q} > 0$
$\mathbf{p} \cdot \mathbf{q} < 0$

Dot product example

- Find the angle α between vectors **p** = [1, 0, 0]^T and **q** = [0, 0, 1]^T.

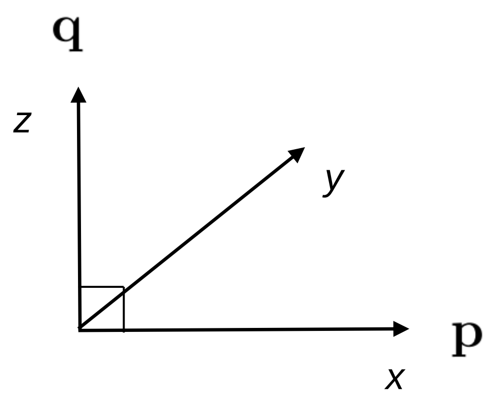
$$\alpha = \cos^{-1}\left(\frac{\mathbf{p}\cdot\mathbf{q}}{||\mathbf{p}||\,||\mathbf{q}||}\right)$$

$$\mathbf{p}\cdot\mathbf{q} = (1)(0) + (0)(0) + (0)(1) = 0$$

$$||\mathbf{p}|| = \sqrt{(1)^2 + (0)^2 + (0)^2} = 1$$

$$||\mathbf{q}|| = \sqrt{(0)^2 + (0)^2 + (1)^2} = 1$$

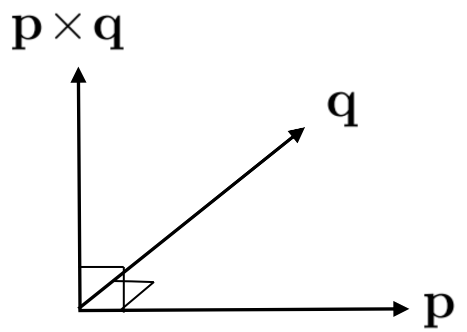
$$\alpha = \cos^{-1}\left(\frac{0}{1\cdot 1}\right) = \cos^{-1}(0) = 90^\circ$$



Cross product

- Cross product
  - The cross product of **p** and **q** returns a new vector orthogonal to both **p** and **q**

$$\mathbf{p} \times \mathbf{q} = \begin{bmatrix} p_y q_z - p_z q_y \\ p_z q_x - p_x q_z \\ p_x q_y - p_y q_x \end{bmatrix}$$

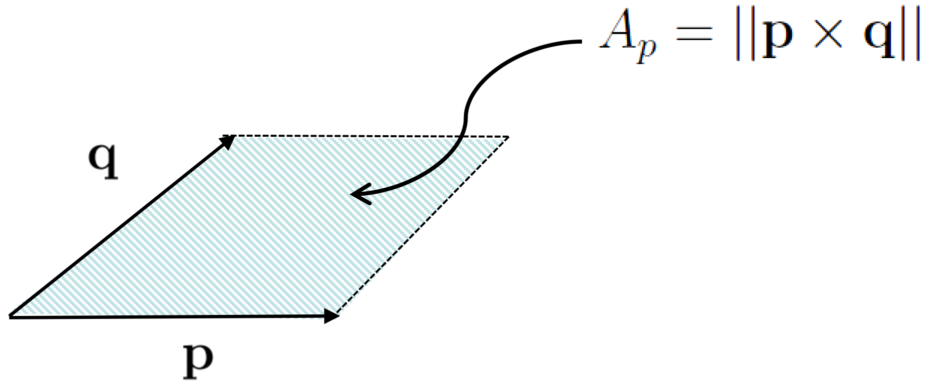


  - Note that the cross product is a *vector*
  - The cross-product obeys the *right hand rule*
    - Using your right hand, point your fingers in the direction of **p**
    - Now curl your fingers towards **q**
    - Your thumb will point in the direction of the cross-product

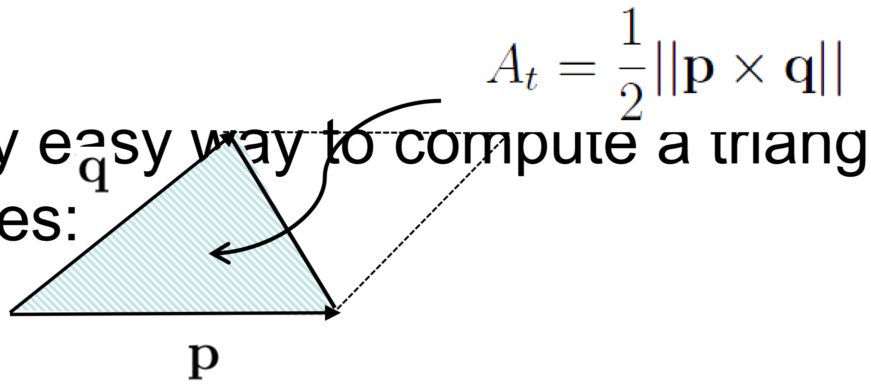
⇨ A positive rotation about the **p** x **q** axis will rotate **p** towards **q**

Area of a triangle

- The magnitude of the cross product is the area of the parallelogram spanned by the two vectors.

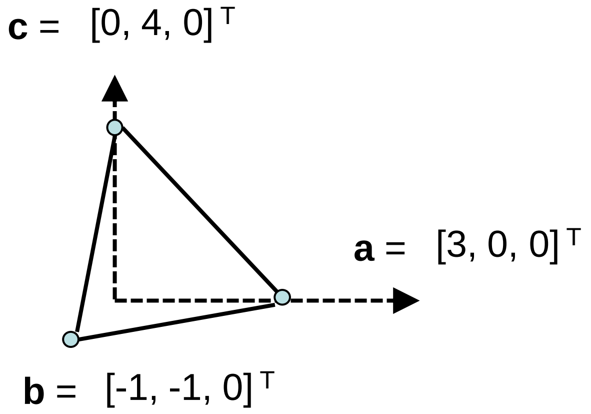


- This gives a really easy way to compute a triangle's area, using vectors along the triangle's edges:



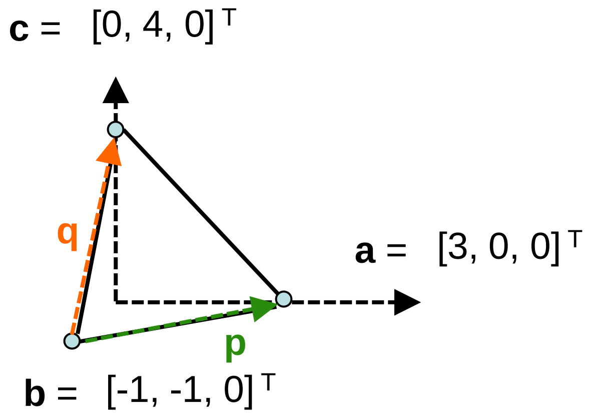
Cross product example

- Find the unit surface normal $\hat{\mathbf{n}}$ of a triangle defined by three points,
  $\mathbf{a} = [3, 0, 0]^T$, $\mathbf{b} = [-1, -1, 0]^T$, $\mathbf{c} = [0, 4, 0]^T$.



$\mathbf{c} = [0, 4, 0]^T$
$\mathbf{a} = [3, 0, 0]^T$
$\mathbf{b} = [-1, -1, 0]^T$

Cross product example



- First, find two vectors on triangle
  - $\mathbf{p} = \mathbf{a} - \mathbf{b} = [4, 1, 0]^T$
  - $\mathbf{q} = \mathbf{c} - \mathbf{b} = [1, 5, 0]^T$

- Next, compute their cross product
  - $\mathbf{n} = \mathbf{p} \times \mathbf{q} \underset{\wedge}{=} [0, 0, 19]^T$

- Finally, normalise
  - $\mathbf{n} = [0, 0, 1]^T$

- Intuitively, the normal should be along the *z*-axis because the triangle lies in the xy plane

Programming vectors using GLM

- GL Mathematics (GLM) is a mathematics library designed to work with modern OpenGL, based on the GL shading language (GLSL) specification
- GLM provides a C++ namespace for mathematics operations. Entities are accessed using the scope operator ::
- It contains vector types like vec2, vec3, and vec4. Common operations (vec3)

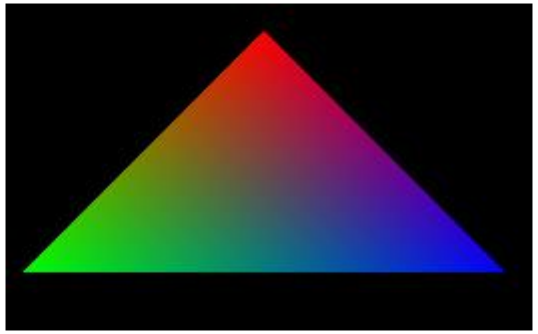
```
#include "include\glm\glm.hpp"
glm::vec3 a = glm::vec3(1, 2, 3);
glm::vec3 b(3, 2, 1);
glm::vec3 c = a - b;
glm::vec3 d = b + 5.0f*a;
glm::vec3 e = glm::cross(c, d);
glm::vec3 f = glm::normalize(e);
float g = glm::dot(c, d);
float h = glm::length(a - b);
float i = glm::distance(a, b);
```

Initialising a vec3

length	distance	dot	cross
normalize	(others)		

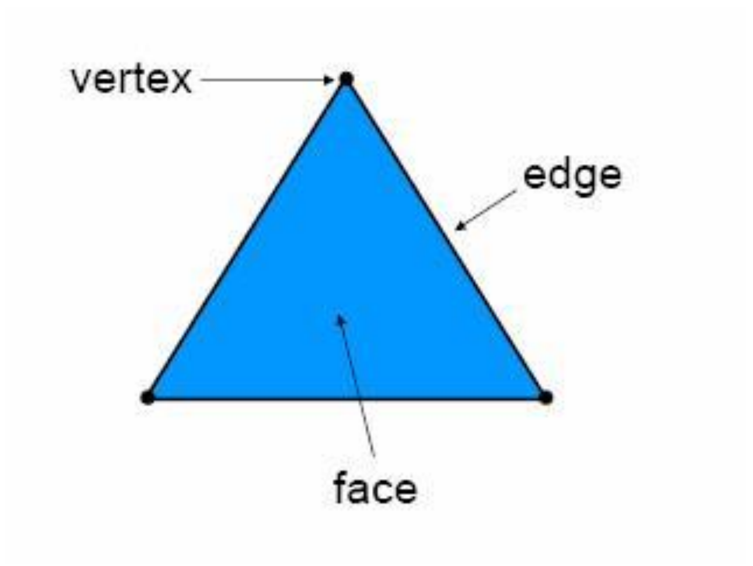
A first OpenGL program

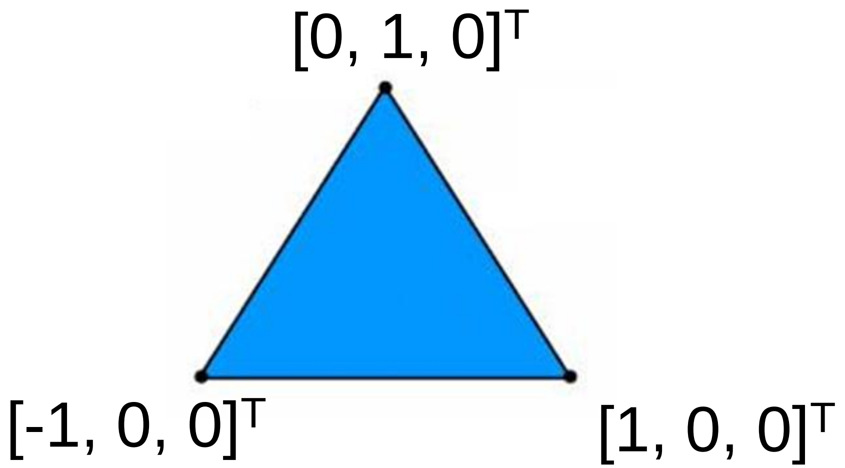
- Triangles are the “bread and butter” of 3D graphics
- We will program OpenGL to render a multi-coloured triangle



- This C++ program will familiarise you with OpenGL 4.0 in a Windows environment

Defining our triangle





- A vertex typically has attributes, like a position, colour, normal, etc.
  - Vertex positions are normally specified as $[x, y]^T$ in 2D and $[x, y, z]^T$ in 3D.
  - Vertex colours are typically in RGB. We'll discuss colour in a later lecture.

OpenGL in Windows

- Microsoft hasn’t contributed to OpenGL since version 1.1. However, much has changed in the 20 years!
- One accesses more recent OpenGL functionality through extensions.
- GLEW (the GL Extension Wrangler) makes this easy, providing access to all the function pointers supported by the d .EW is available at

  http://glew.sourceforge.net/

- One must initialise GLEW using a temporary OpenGL 1.1 rendering context. Once GLEW is initialised, this rendering context is deleted and one can create a more modern rendering context.

Pixel formats

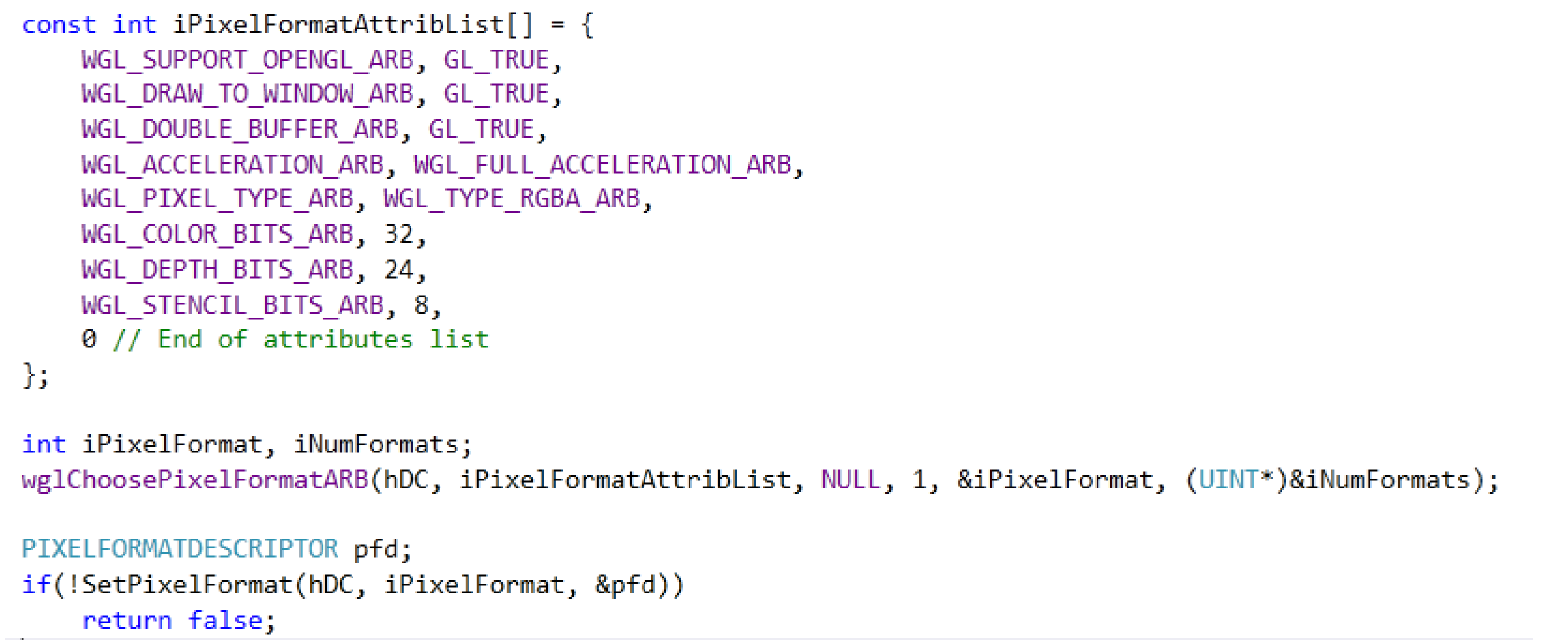
- Before rendering to a window, one must configure the window according to the rendering requirements. For example, setting
    - The number of colour buffers (single or double?) and their format
    - If a depth buffer is required and its format
    - If a stencil buffer is required
    - Version of OpenGL
- Once these parameters are set, they cannot be changed. Reconfiguring will require destroying the window and creating a new window with the desired format.
- WGL uses pixel formats to encapsulate all of this information

Pixel formats

- Specifying the pixel format is done with the function

`wglChoosePixelFormatARB(HDC hdc, const int *piAttribIList, const float    *pfAttribFList, UINT nMaxFomats, const int *piFormats, UINT *nNumFormats);`

  - `piAttribIList` is a null-terminated array of integer attribute pairs
- OpenGL will return a pixel format that is the best match in `iPixelFormat`
- The example below finds a pixel format for a double-buffered, hardware accelerated window with 32 bits colour per pixel, a 24 bit depth buffer, and an 8 bit stencil buffer.



```
const int iPixelFormatAttribList[] = {
    WGL_SUPPORT_OPENGL_ARB, GL_TRUE,
    WGL_DRAW_TO_WINDOW_ARB, GL_TRUE,
    WGL_DOUBLE_BUFFER_ARB, GL_TRUE,
    WGL_ACCELERATION_ARB, WGL_FULL_ACCELERATION_ARB,
    WGL_PIXEL_TYPE_ARB, WGL_TYPE_RGBA_ARB,
    WGL_COLOR_BITS_ARB, 32,
    WGL_DEPTH_BITS_ARB, 24,
    WGL_STENCIL_BITS_ARB, 8,
    0 // End of attributes list
};

int iPixelFormat, iNumFormats;
wglChoosePixelFormatARB(hDC, iPixelFormatAttribList, NULL, 1, &iPixelFormat, (UINT*)&iNumFormats);

PIXELFORMATDESCRIPTOR pfd;
if(!SetPixelFormat(hDC, iPixelFormat, &pfd))
    return false;
```

Creating the rendering context

- The rendering context is created using the command

```
wglCreateContextAttribsARB(HDC hdc, HGLRC hShareContext, const int *attribList)
```

- The `attribList` is an array of rendering context attribute pairs. With this, one can select the version of OpenGL. Note however, OpenGL may create any context that is backwards compatible with the version requested.
- Not requesting a specific version often creates a context with the latest version -- however, this behaviour depends on the vendor.
- The example below creates an core context and makes it current.

```
int iContextAttribs[] =
{
   WGL_CONTEXT_MAJOR_VERSION_ARB, iMajorVersion,
   WGL_CONTEXT_MINOR_VERSION_ARB, iMinorVersion,
   WGL_CONTEXT_PROFILE_MASK_ARB,
WGL_CONTEXT_CORE_PROFILE_BIT_ARB,
   0 // End of attributes list
};

m_hrc = wglCreateContextAttribsARB(m_hdc, 0,
iContextAttribs);

if (m_hrc) wglMakeCurrent(m_hdc, m_hrc);
else bError = true;
```

Creating the rendering context

- The attribute WGL_CONTEXT_PROFILE_MASK_ARB is followed by a flag containing either
  - WGL_CONTEXT_CORE_PROFILE_BIT_ARB: **Creates a core context** with no deprecated functionality available. It is generally good programming practice to set this flag and essential for the work we will do in this module.
  - WGL_CONTEXT_COMPATIBLITY_PROFILE_BIT_ARB: **Creates a compatibility context** that is backwards compatible with all older versions of OpenGL. All deprecrated functions available. This context may run slower due to additional state and functionality that needs to be tracked.

Identifying the OpenGL version

- A rendering context can only be created up to the version of OpenGL supported by the OpenGL driver.

- A program *glewInfo* (part of the GLEW distribution) can provide all the OpenGL capabilities of your hardware. After running it, the program will output a text file describing all the OpenGL versions and extensions supported by the GPU.

- To determine the OpenGL version at runtime, call
  - `const GLubyte *verString = glGetString(GL_VERSION);`

- If you're working on your own computer, **check which version of OpenGL it supports**