# MSc Computer Games and Entertainment

Maths & Graphics Unit 2011/12

Lecturer: Gareth Edwards

# Part 2:

## Mathematics for Computer Graphics

## Let’s take a closer look at the Geometry and Rasterizer Stages

## So:

The programmer in an Application stores via API calls, graphic primitives on a GPU, then performs via further API calls “geometrical” operations in a specific order such that when these graphic primitives are rendered via even more API calls, they are transformed, projected, and rendered using the GPU Rasterizer into a GPU image buffer.

- The programmer performs these actions on the GPU using an API
- The GEOMETRY stage does per vertex operations
- The Rasterizer stage does per pixel operations

## Coordinate Systems

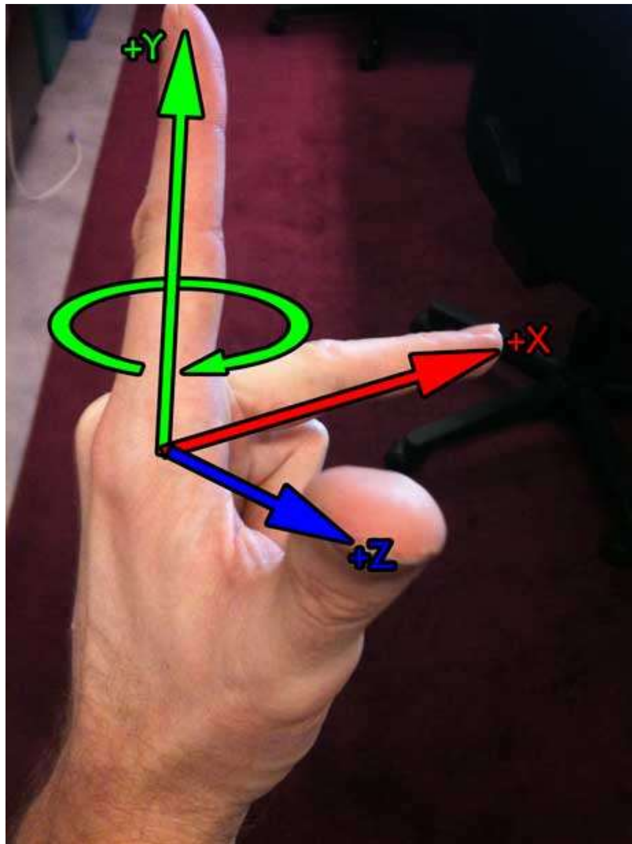
Before we can talk about transformations, we must make a formal definition of what our coordinate system is.

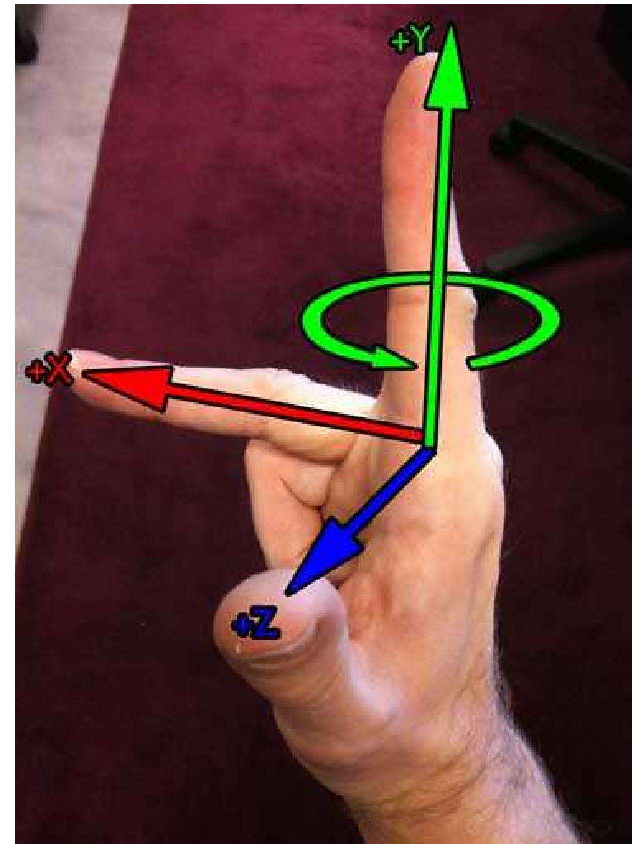
Throughout this course, I will be using a left-handed coordinate system. This is the default coordinate system used by DirectX.

The default coordinate system used by OpenGL is a right-handed coordinate system!

## Coordinate Systems

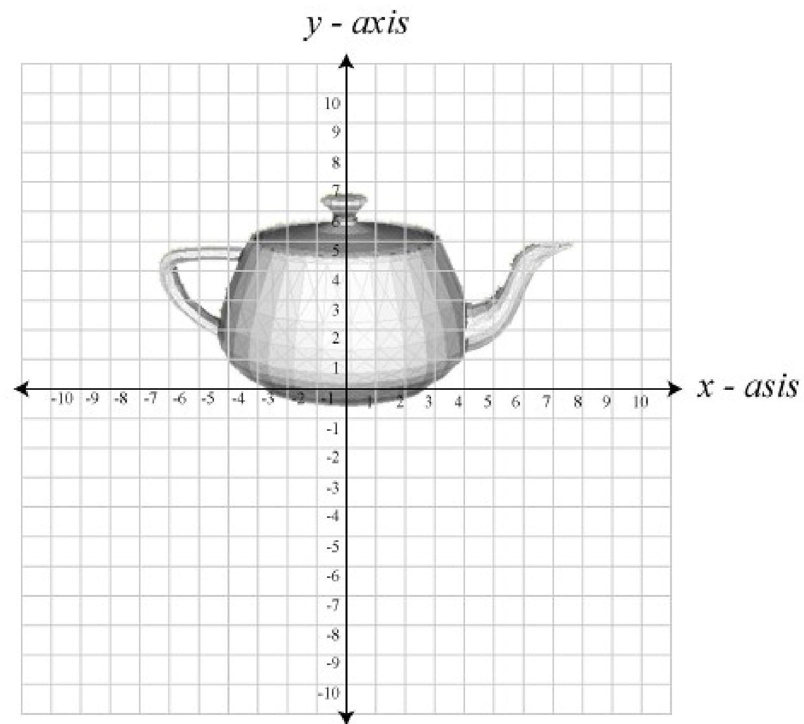
Left or right handed coordinate systems!



## Model Space

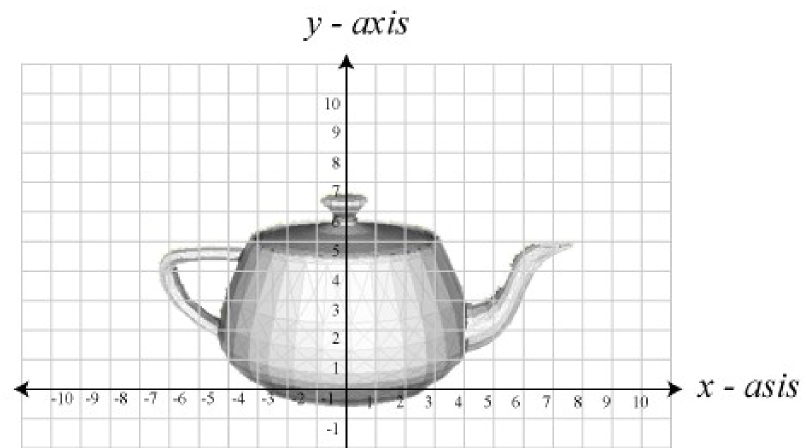
The image below shows an example of an object in object space. As you can see from the image, the object is placed at it's relative origin.



## Model Space

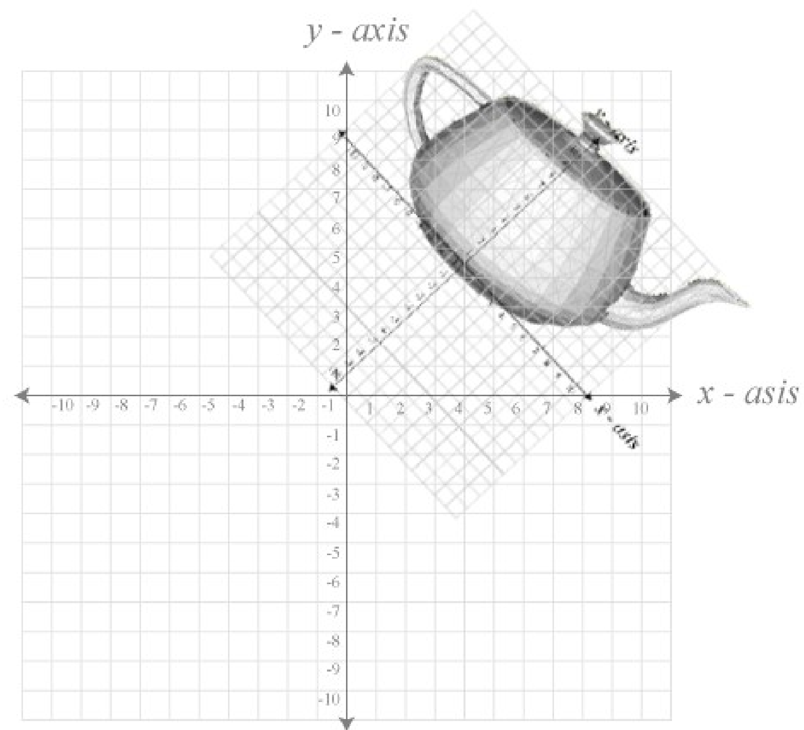
Originally, an object is in "**model space**" – sometimes known as "local space" or "object space".

In "model space" an object's vertices are expressed relative to the object that they describe. That is, the way the artist intended for them to be displayed if you didn't move the object from the origin.



## World Space

A programmer within an Application using API calls, then transforms (moves, rotates, scales) geometrical objects into “**world space**”.



## World Space

In the example shown:

- A teapot is defined with origin at (0,0,0)
- Using Translate and Rotate API calls the teapot is rotated and translated

These transforms are performed per vertex with concatenated 4x4 matrix multiplication.

- The parameters passed to the GPU via the API calls can update the contents of the transform matrices over time
- Thus an objects can be animated

## Combining Coordinate Spaces

- In 3D computer graphics, coordinate spaces are described using a homogeneous coordinate system. A homogeneous coordinate system allows us to represent all of our affine transformations (translation, rotation, scale, and perspective projection) in a similar way so they can easily be combined into a single representation.

- Any number coordinate spaces can be combined using matrix multiplication which results in a single matrix that can be applied to all the vertices of an object.

## Combining Coordinate Spaces

- Even multiple world coordinate spaces can be combined in order to derive a final coordinate space that describes the location of all of our vertices in an object.

- This is useful for nested coordinate spaces where the position of an object is expressed relative to a “parent” object. When the parent object’s world transform is changed, the transform of the child object is also changed implicitly.

- Using this method, complex scenes can be constructed from several smaller scenes and placed into the larger scene to create a complete world.

## View Space

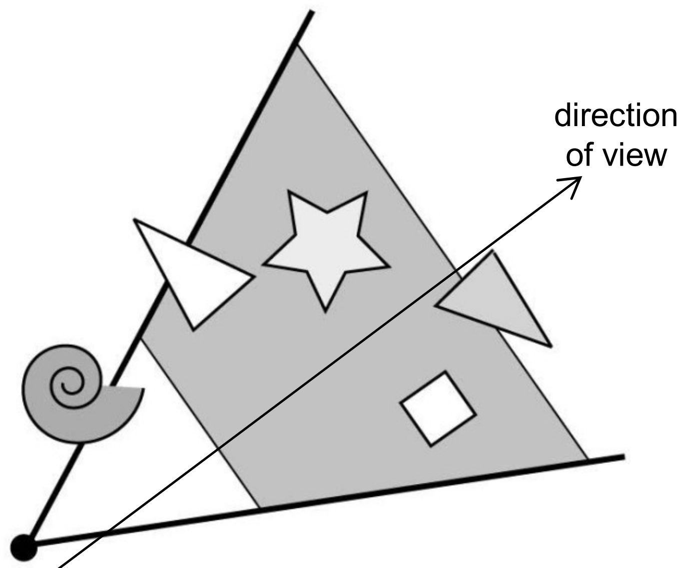
“View space” is also known as “camera space”.

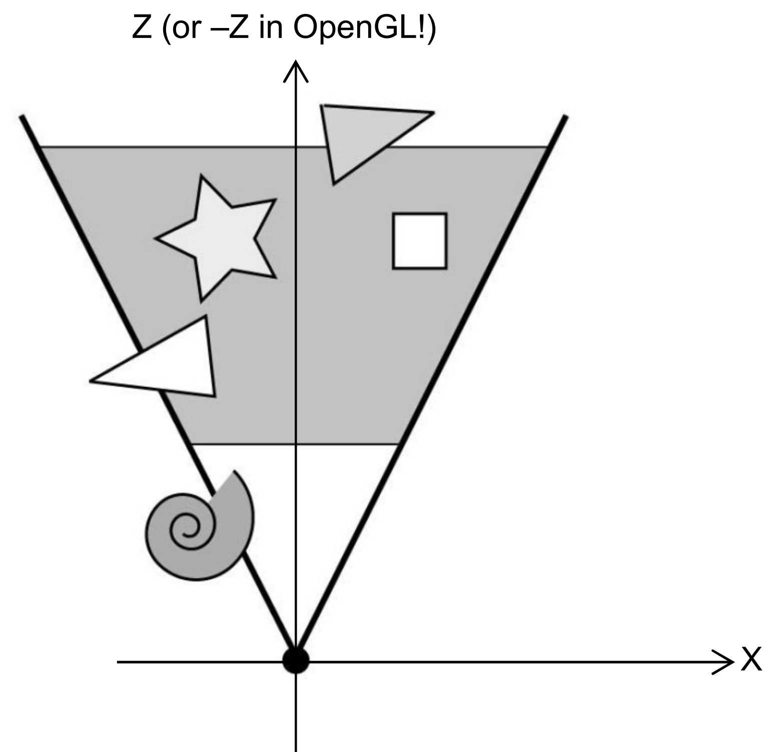
- “view space” is the coordinate space that is associated with the observer.
- “view space” is considered to be the origin and orientation of what we are looking at.
- The viewer’s coordinate axis usually assume the positive axis points right, the positive axis points up, and in a left-handed coordinate system, the positive axis points forward, or at the scene. In a right-handed coordinate system, the axis is reversed, so the negative axis points forward, or at the object in our scene. A more common name for these axes are the “Right”, “Up”, and “At” axes.

## View Space

To transform a 3D scene into “view space”:

- Apply inverse transform to objects, so that viewer looks down z-axis

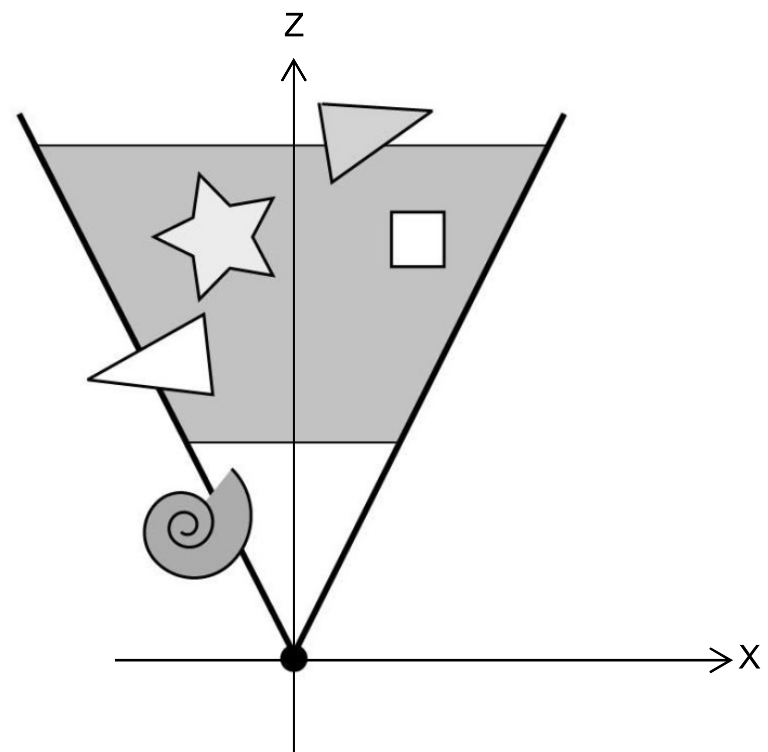


## View Space

The camera space transformation together with the projection transformation is useful for answering such questions as:

- Is an object completely in view, partially in view, or not in view at all?
- Is one object closer to the camera than the other?
- Is an object directly in front of, above, below, to the left, or to the right of the camera?



## Lighting and Shading

The Rasterizer uses lighting, materials and textures set up in the Geometry stage to compute “illumination” at vertices and also per-pixel

- This process attempts to mimic how light in the “real world” behaves
  - It is hard to uses empirical models, and GPU based lighting and shading is an approximation
- Much more about this in later lectures!

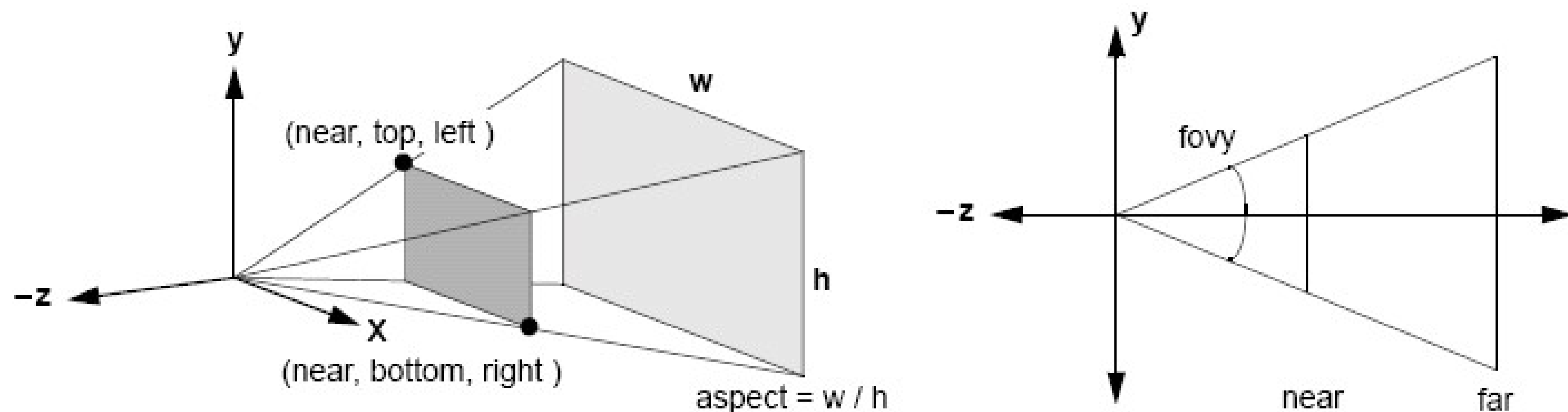
## Projection

There are two major ways to do it

- Orthogonal (few applications: e.g. architecture)
- Perspective (most often used)

This mimics how humans perceive the world, i.e., an objects apparent size decreases with distance.

- Normally projection is based on a pinhole camera analog



## Projection Matrix

Projection is accomplished with matrix multiplication using the Projection matrix.

If a view volume is defined as:

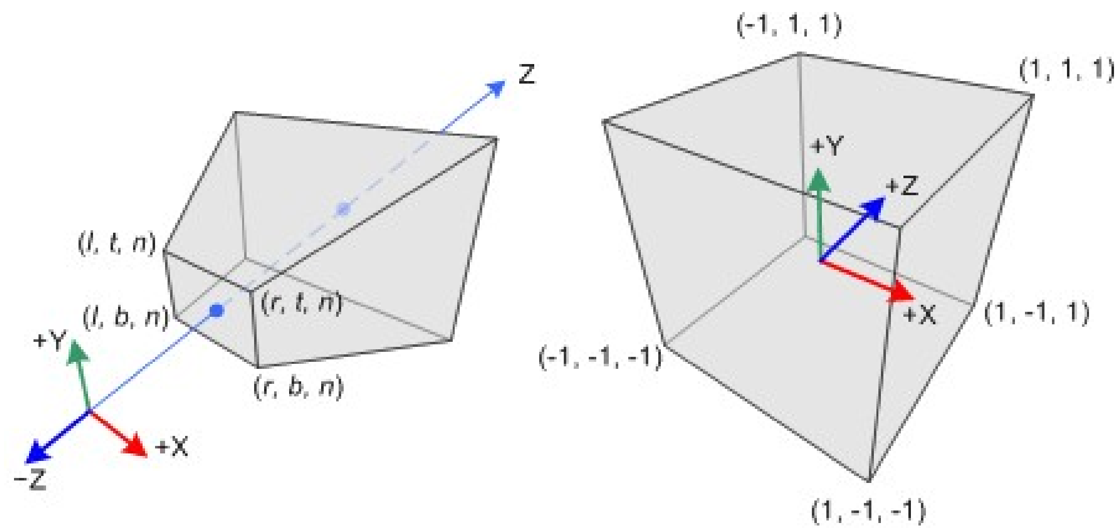
- Sw-screen window width in camera space in near clipping plane
- Sh-screen window height in camera space in near clipping plane
- Zn-distance to the near clipping plane along Z axes in camera space
- Zf-distance to the far clipping plane along Z axes in camera space

Then a perspective projection matrix could be written as follows:

$$M_{proj} = \begin{pmatrix} \frac{2 * Zn}{Sw} & 0 & 0 & 0 \\ 0 & \frac{2 * Zn}{Sh} & 0 & 0 \\ 0 & 0 & \frac{Zf}{Zf - Zn} & 1 \\ 0 & 0 & \frac{-Zf * Zn}{Zf - Zn} & 0 \end{pmatrix}$$

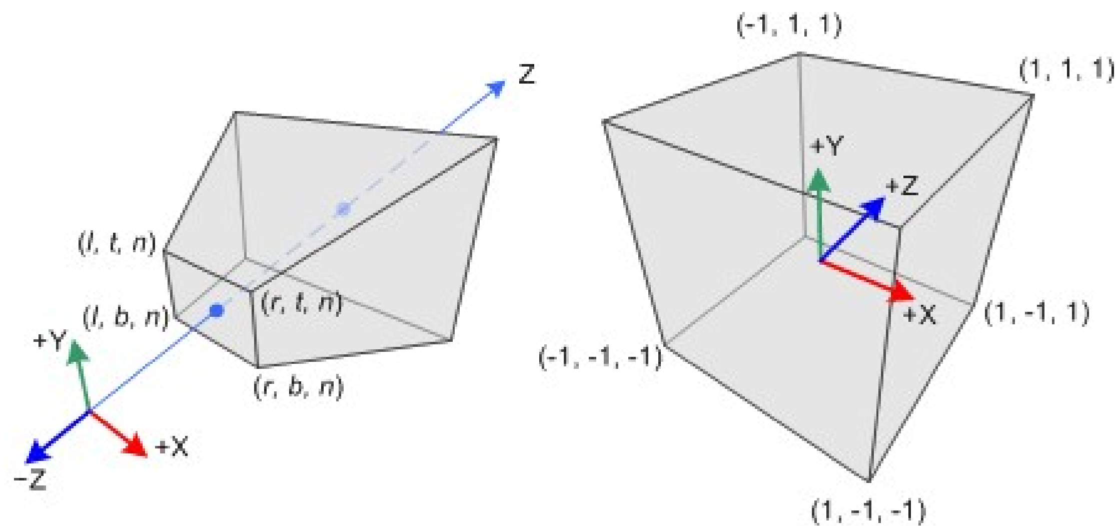
## Image Space or Normalised Device Coordinates (NDC)

In perspective projection, a 3D point in a truncated pyramid frustum ("view space" coordinates) is mapped to a cube (NDC); the x-coordinate from [l, r] to [-1, 1], the y-coordinate from [b, t] to [-1, 1] and the z-coordinate from [n, f] to [-1, 1].



## Image Space or Normalised Device Coordinates (NDC)

Note that the DirectX eye and the NDC coordinates are defined in left-handed coordinate system. That is, the camera at the origin is looking along Z axis in both. This avoid a conceptual problem in OpenGL where this is not the case, and where the near and far plane values must be specified as positive values!



## Clipping and Screen Mapping

Clipping and screen mapping provide the final stages that complete the 3D pipeline.

- Projected graphic primitives are clipped to this cube
- This screen mapping, scales and translates the square so that it ends up being mapped in to a rendering window (e.g. from square to rectangle)
- These “screen space coordinates” together with Z (depth) are sent to the Rasterizer stage

## Summary of the Geometry Stage

The Geometry stage:

- Sets up lighting and materials
- Sets up the virtual camera
- Transforms from “model space” into “world space”
- Transforms “world space” into “view” (virtual camera) space
- Performs projection into “image space”
- Clips within “image space”
- Maps to screen

## Summary of the Geometry Stage

In the transformational pipeline:

If a vertex in the model coordinate is given by Pm = (Xm, Ym, Zm, 1), then the transformations shown in the following figure are applied to compute screen coordinates Ps = (Xs, Ys, Zs, Ws).

Pm = (Xm, Ym, Zm, 1) ⇨

1. Model space
2. World space
3. View space
4. Projection space
5. Clipping space
6. Homogenous space

⇨ Ps = (Xs, Ys, Zs, Ws)

## The Rasterizer Stage

Scan conversion:

- Find out which pixels are inside the primitive

Texturing:

- Put images on triangles

Interpolation over triangle

Z-Buffering:

- Make sure that what is visible from the camera really is displayed

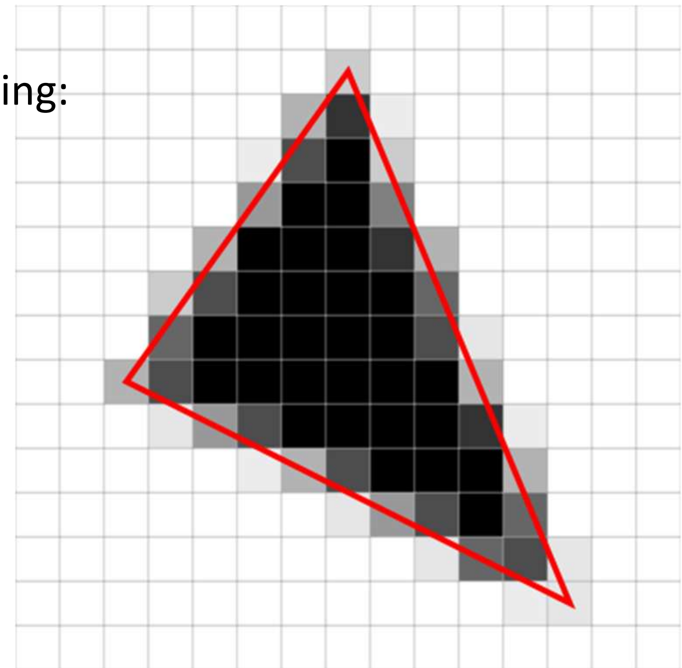
Double buffering (due to monitors refresh rate)

And much more…

## Scan conversion

For each Triangle passed from GEOMETRY to the RASTERIZER:

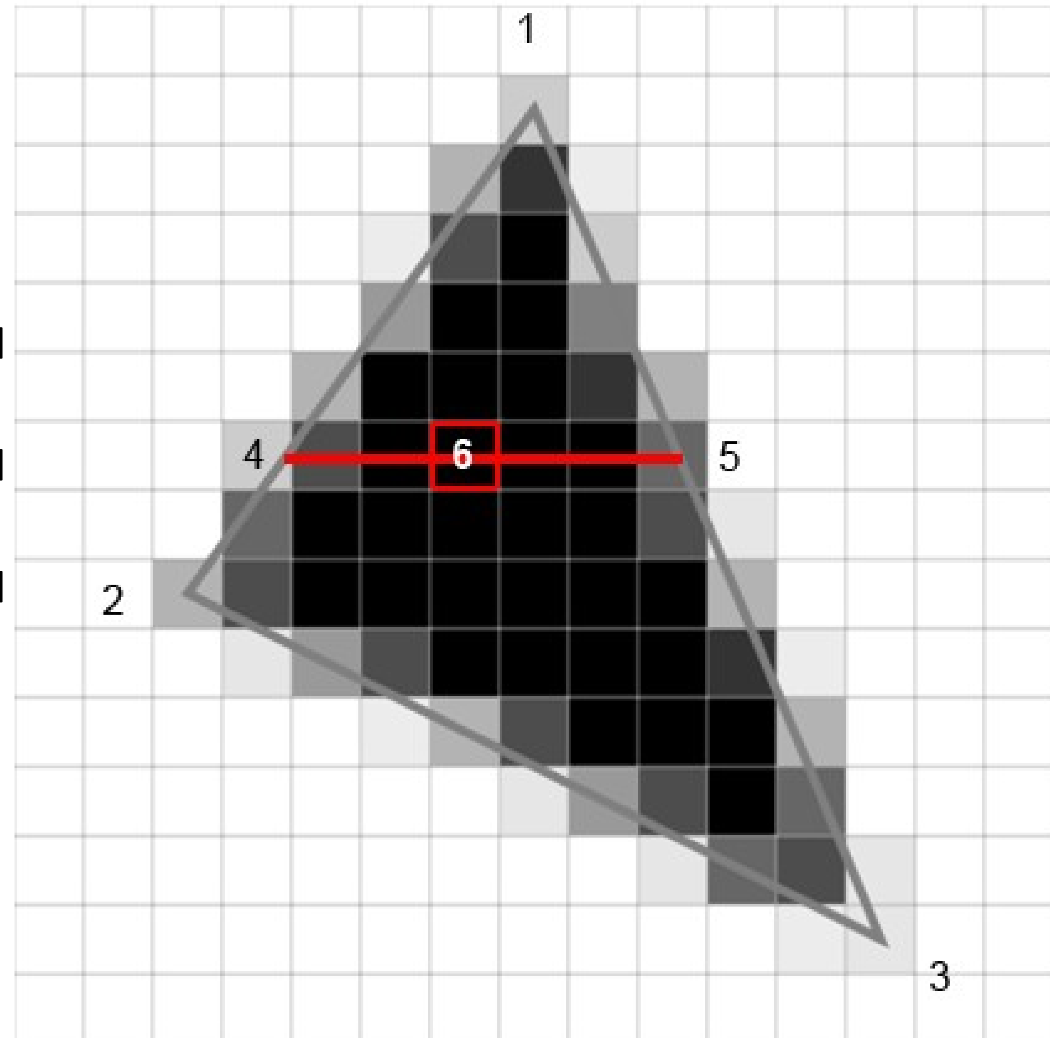
- Find pixels inside the triangle
  - or on a line, or on a point
- Do per pixel operations on these pixels, including:
  - Interpolation
  - Texturing
  - Z-Buffering
  - And more…



## Interpolation

Calculate pixel values over a triangle using Gourard shading.

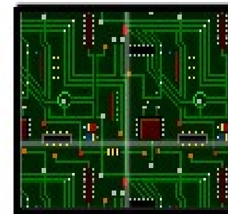
- Interpolate between 1 and 2 to get 4
- Interpolate between 1 and 3 to get 5
- Interpolate between 4 and 5 to get 6



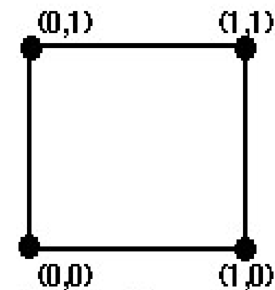
# Texturing

Uses and other applications include:

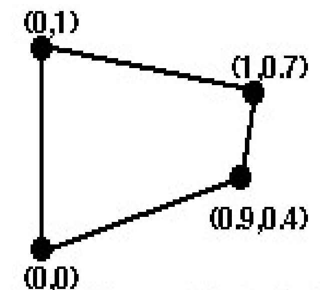
- More realism
- Bump mapping
- Pseudo reflections
- Store or “bake” lighting
- Almost infinitely many uses



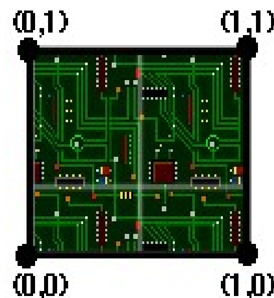
The texture



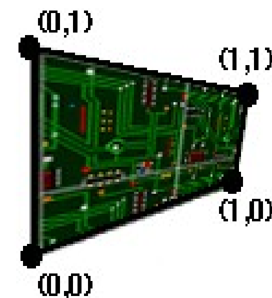

Square with geometric coordinates shown.



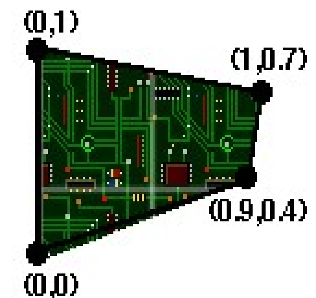

Irregular quadrilateral with geometric coordinates shown.




Square with texture coordinates shown.




Irregular quadrilateral with texture coordinates shown. Note the distortion.




Irregular quadrilateral with texture coordinates matching geometric coordinates. Note the lack of distortion.

## Z-Buffering

Typically graphics hardware – the GPU - is pretty stupid.

It ”just” draws triangles.

- However, a triangle that is totally covered by another more closely located triangle should not be visible.
- Assuming two equally large triangles at different depths:

## Z-Buffering

It would be nice to avoid sorting as it would require substantial computationally overheads.

- The Z-Buffer (aka depth buffer) solves this.

- Idea:
  - Store z (depth) at each pixel
  - When scan converting a triangle, compute z at each pixel on triangle
  - Compare triangle's z to Z- Buffer Z value
  - If triangle's z is smaller, then replace Z-Buffer and color buffer
  - Else do nothing

- Can render in any order (like a spatial lookup table)

## Double buffering

The monitor displays one image at a time.

But if a real-time application did this then:

- Rendered primitives pop up
- Even worse, we often clear the screen before generating a new image

A better solution is ”double buffering”

**And, in recent years:**

Programmable shading has become a hot topic

- Vertex shaders
- Pixel shaders
- Adds more control and much more possibilities for the programmer

**And more recently:**

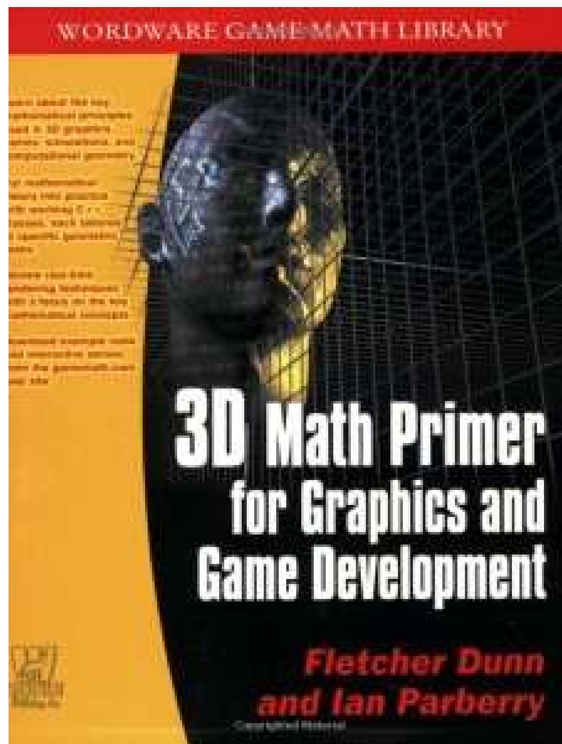
GPGPU - General Purpose computing on GPU’s

# Course Books

Freely available for download from the web
and also the course website
( http://grad.gold.ac.uk: VGS ▶ MSc CGE ▶ Files ▶ Maths Books)

## 3D Math Primer for Graphics and Game Development

Fletcher Dunn &Ian Parberry (2002).



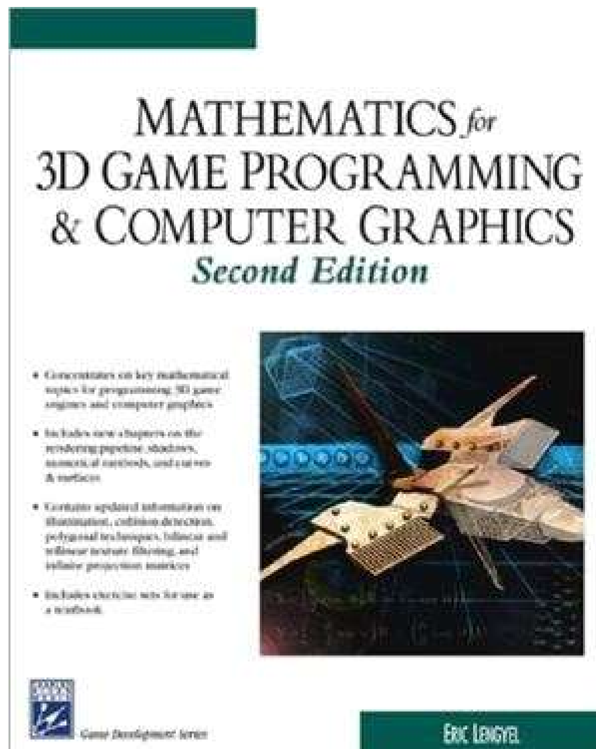

This book covers fundamental 3D fundamental maths concepts that are especially useful for computer game developers and programmers.

The authors discuss the mathematical theory in detail and then provide the geometric interpretation necessary to make 3D maths intuitive.

Working C++ classes illustrate how to put the techniques into practice, and exercises at the end of each chapter help to reinforce the concepts..

## Mathematics for 3D Game Programming and Computer Graphics

Eric Lengyel (2003)




This completely updated second edition illustrates the mathematical concepts that a game programmer would need to develop a professional-quality 3D engine.

Although the book is geared toward applications in game development, many of the topics appeal to general interests in 3D graphics. It starts at a fairly basic level in areas such as vector geometry and linear algebra, and then progresses to more advanced topics in 3D game programming such as illumination and visibility determination.

Particular attention is given to derivations of key results, ensuring that the reader is not forced to endure gaps in the theory.

The book assumes a working knowledge of trigonometry and calculus, but also includes sections that review the important tools used from these disciplines, such as trigonometric identities, differential equations, and Taylor series.

# Reading and work for next week!

## Reading and work to be undertaken for next week....

Please:

- Skim through the two course books.
- Read through today's lecture (will be available before weekend) .
- Compile and run RTVS_Lite (distributed at today’s lecture).

And:

- All future course work and assignments will require mastery of the concepts, ideas and code discussed and examined today!!!
- Next week – VECTORS.

# End of Part 2:

Mathematics for
Computer Graphics