University of BRISTOL

COMS 30115

Clipping and Culling

Carl Henrik Ek - carlhenrik.ek@bristol.ac.uk

March 6th, 2017

http://www.carlhenrik.com

Introduction

Last time

- Mappings
  - Not just textures
- Optimisations
  - SSE Instructions
  - Fixed-Point-Math

Today

- Clipping and Culling

The Book

- Scratchapixel URL
- Blinn and Newell URL
- Fabien Sanglard Webpage URL
- Keneth Joy notes on Clipping URL

Clipping

Our Rasterisation Engine

- We know how to transform geometry
- We know how to project things from 3D space to screen space
- We know how to draw 3D data by interpolation in screen space
- Now we need to figure out what to draw

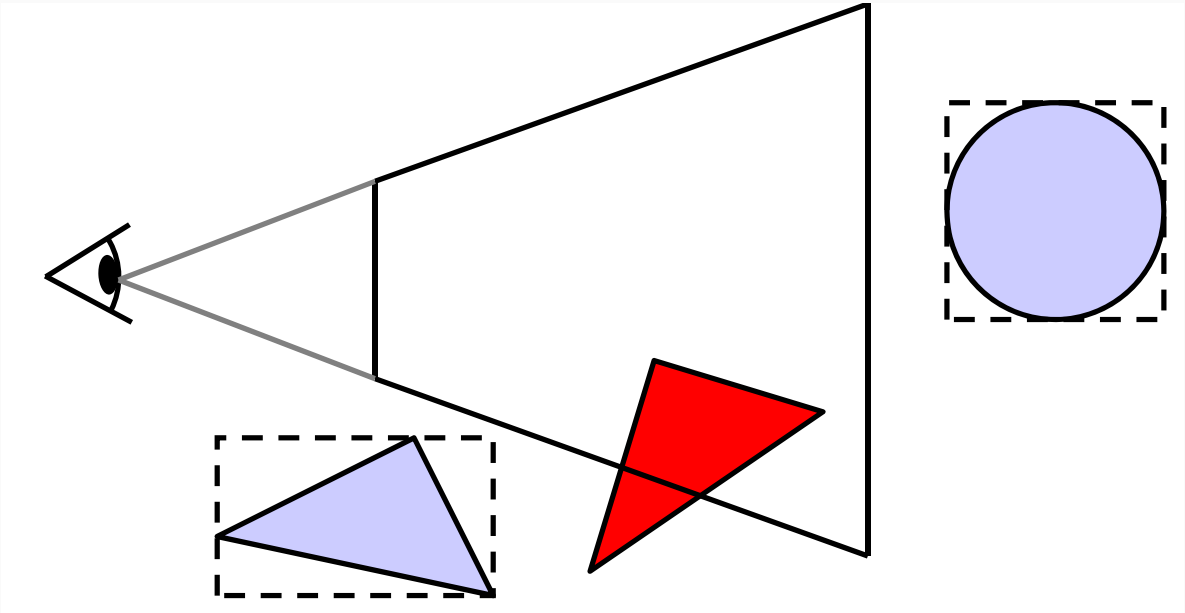
Pipeline

- ☒ Transform world
- ☐ Clip geometry to view Frustrum
- ☒ Project vertices
- ☒ Interpolate depth across polygons
- ☐ Perform depth culling
- ☒ Interpolate shading/textures etc.
- ☒ Perform pixelshading
- ☒ Double buffer
- repeat

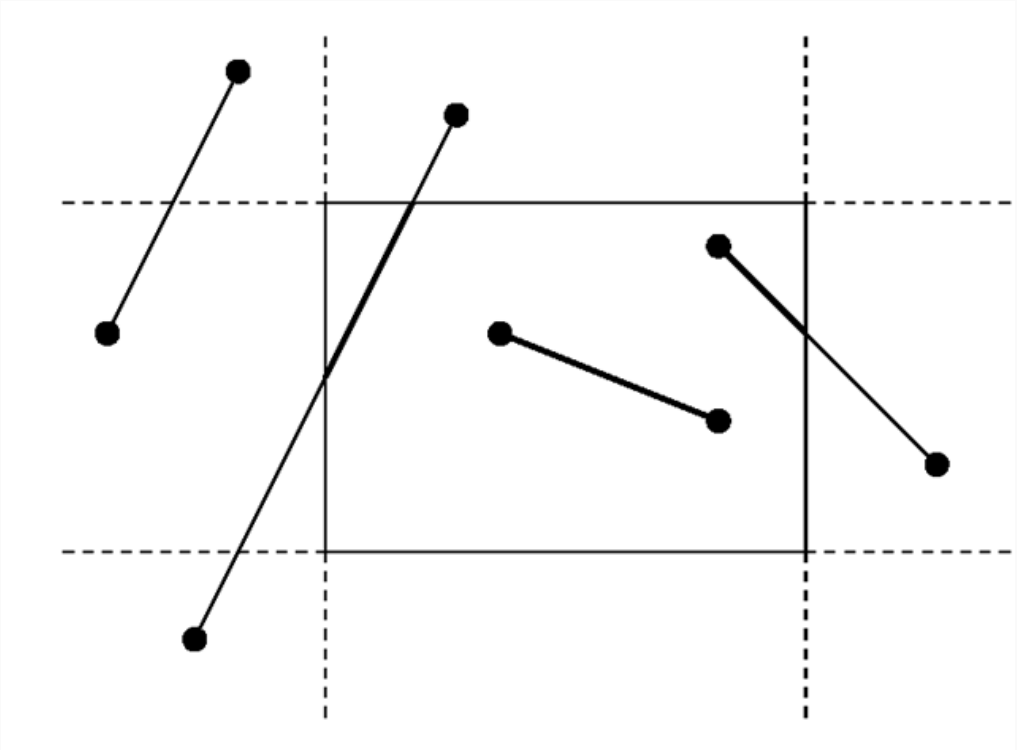
Clipping & Culling

- Drawing each polygon expensive
- Remove elements that we do not need to draw early in the pipeline to save computations
- Culling: remove whole primitive
  - back-face culling, occlusion culling, etc.
- Clipping: remove part of primitive

Clipping & Culling

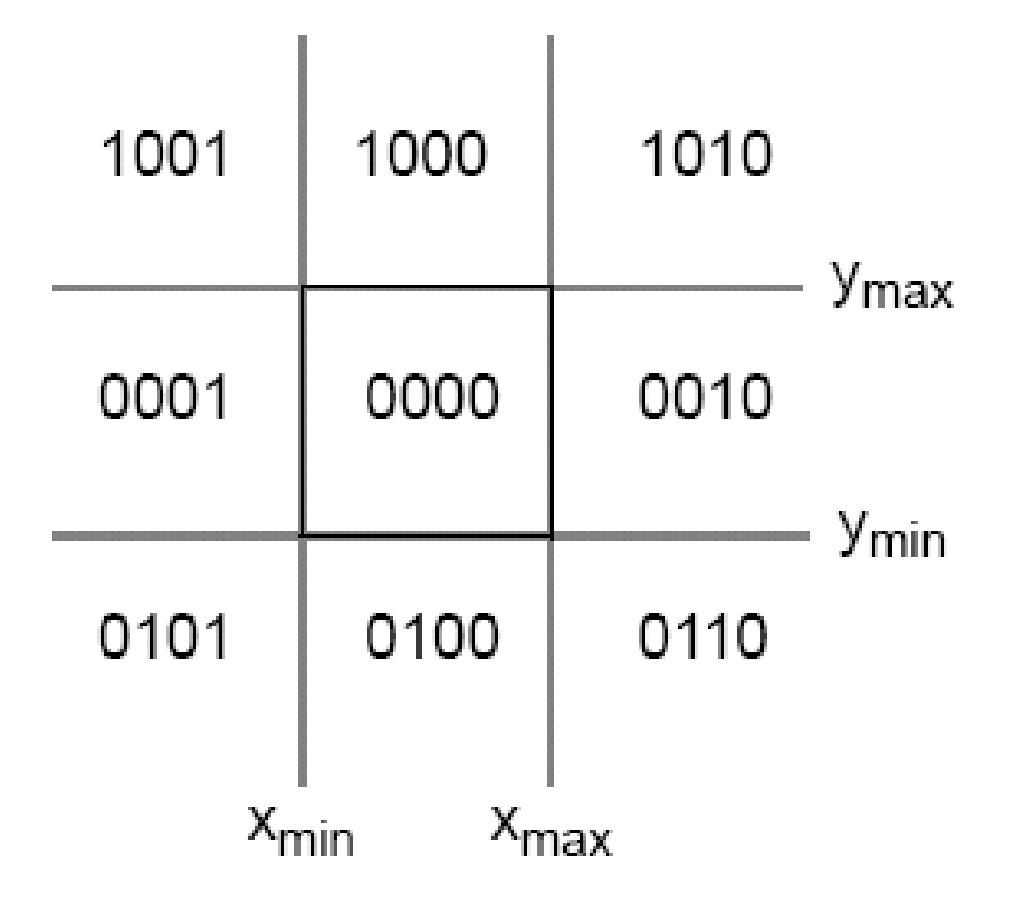


Line Clipping[1]



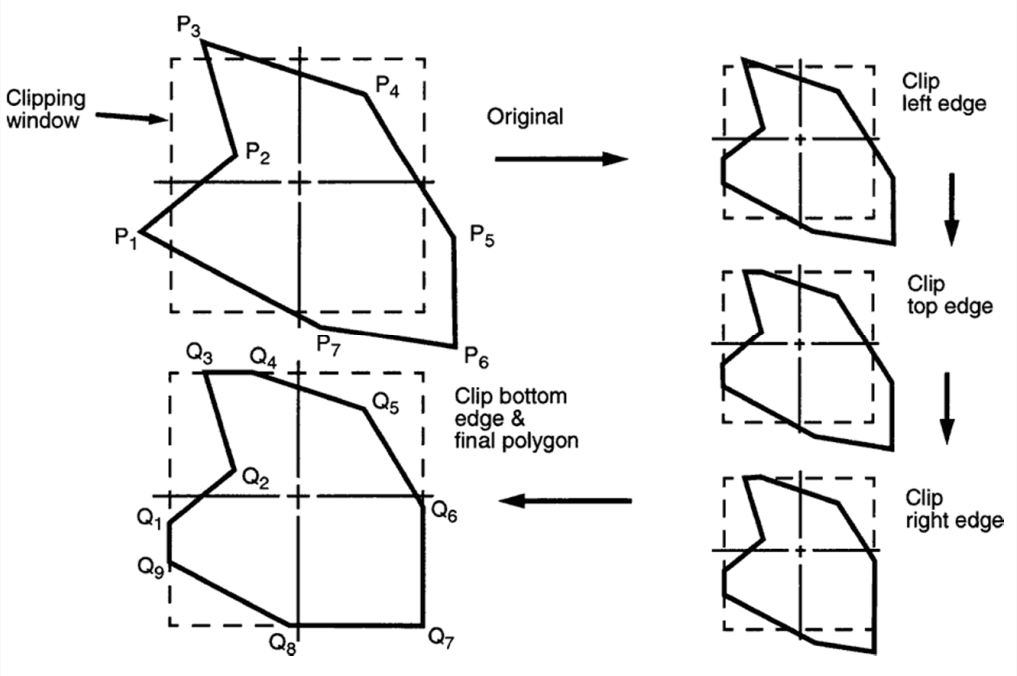
[1] https://en.wikipedia.org/wiki/Cohen%E2%80%93Sutherland_algorithm

Line Clipping[1]



1. Compute `outcode` for each end-point
2. Reject and accept trivial cases quickly
3. Clip remaining

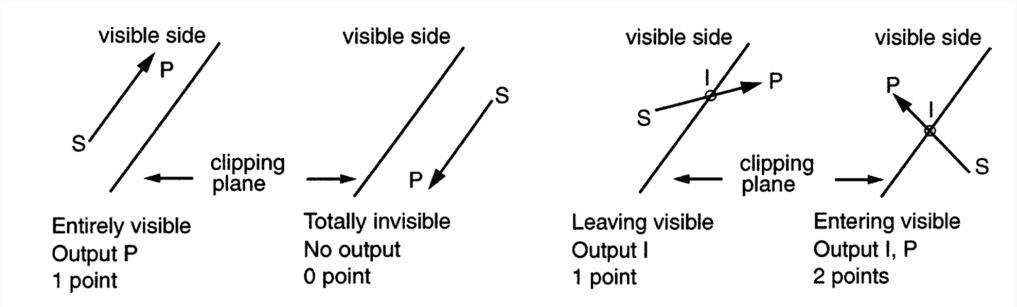
Polygon Clipping[2]



Sutherland-Hodgman Polygon clipping

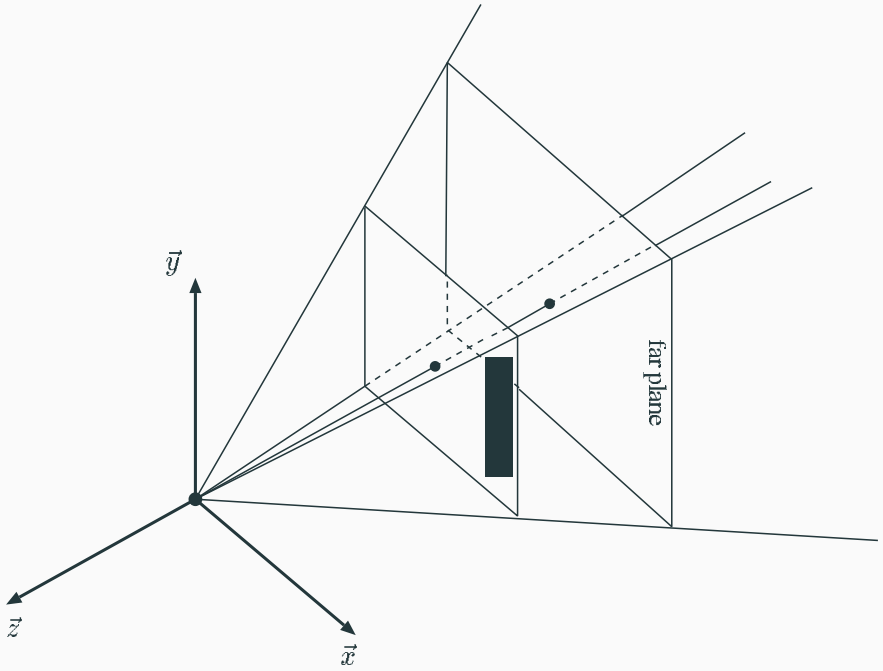
[2] https://en.wikipedia.org/wiki/Sutherland%E2%80%93Hodgman_algorithm

Polygon Clipping[2]



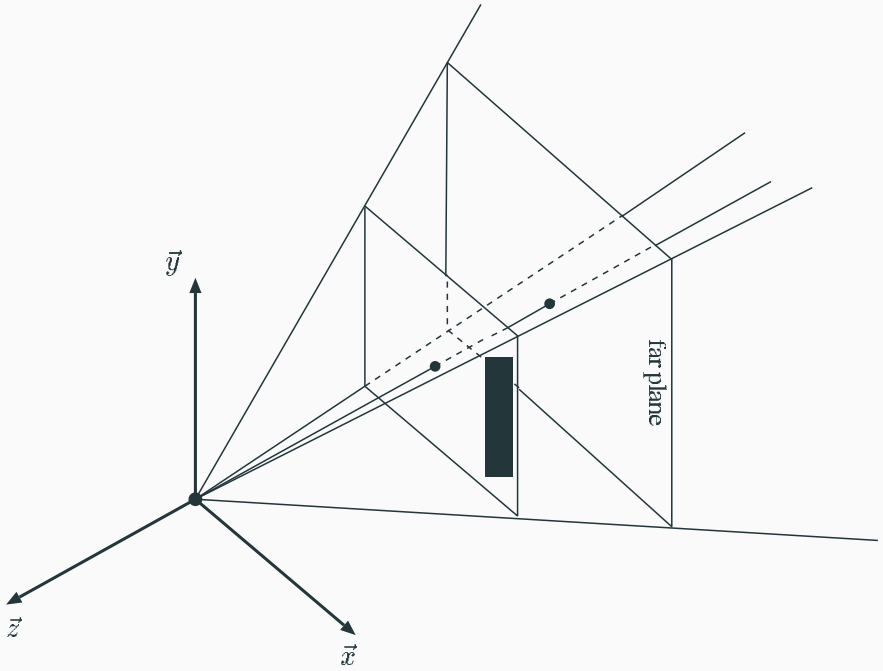
[2] https://en.wikipedia.org/wiki/Sutherland%E2%80%93Hodgman_algorithm

3D Clipping



- We can simply extend 2D cases to 3D
- Compute view frustrum and clip against planes
- Annoying with non-axis aligned planes

3D Clipping



- We can simply extend 2D cases to 3D
- Compute view frustrum and clip against planes
- Annoying with non-axis aligned planes
- Clip in homogenous coordinates

Homogenous Coordinates

$$u = \frac{f}{z} \cdot x$$

$$v = \frac{f}{z} \cdot y$$

$$\begin{bmatrix} u \\ v \\ f \\ 1 \end{bmatrix} = \frac{f}{z} \begin{bmatrix} x \\ y \\ z \\ \frac{z}{f} \end{bmatrix}$$

$$\begin{bmatrix} x \\ y \\ z \\ \frac{z}{f} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{1}{f} & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Homogenous Clipping

- We can add a single coordinate w to each point

$$[x, y, z, w]^{\mathrm{T}}$$

- the process of homogenisation is to make $w = 1$ which corresponds to projecting $[x, y, z, w]^{\mathrm{T}}$ to its corresponding point $[\frac{1}{w}x, \frac{1}{w}y, \frac{1}{w}z, 1]^{\mathrm{T}}$ which is a point in 3D space
- This means $[x, y, z, 1]^{\mathrm{T}}$ and $[3x, 3y, 3z, 3]^{\mathrm{T}}$ corresponds to the same point in 3D space

Homogenous Clipping

- To go from homogenous coordinates we project from a 4D space to 3D plane
- To go from world space to screen space we project from a 3D space to a 2D plane

$$[x, y, z, 1]^{\mathrm{T}} \rightarrow [\frac{f}{z}x, \frac{f}{z}y, \frac{f}{z}z, 1]^{\mathrm{T}}$$

- We can write the set of *all* coordinates that coresponds to a screen coordinate with a *single* homogenous coordinate

$$[x, y, z, \frac{z}{f}]^{\mathrm{T}}$$

- In this space clipping is easy, $x > |w \cdot x_{\max}|$ are all points that should be clipped in x-plane

Homogenous Clipping

1. Map from world space to clip space

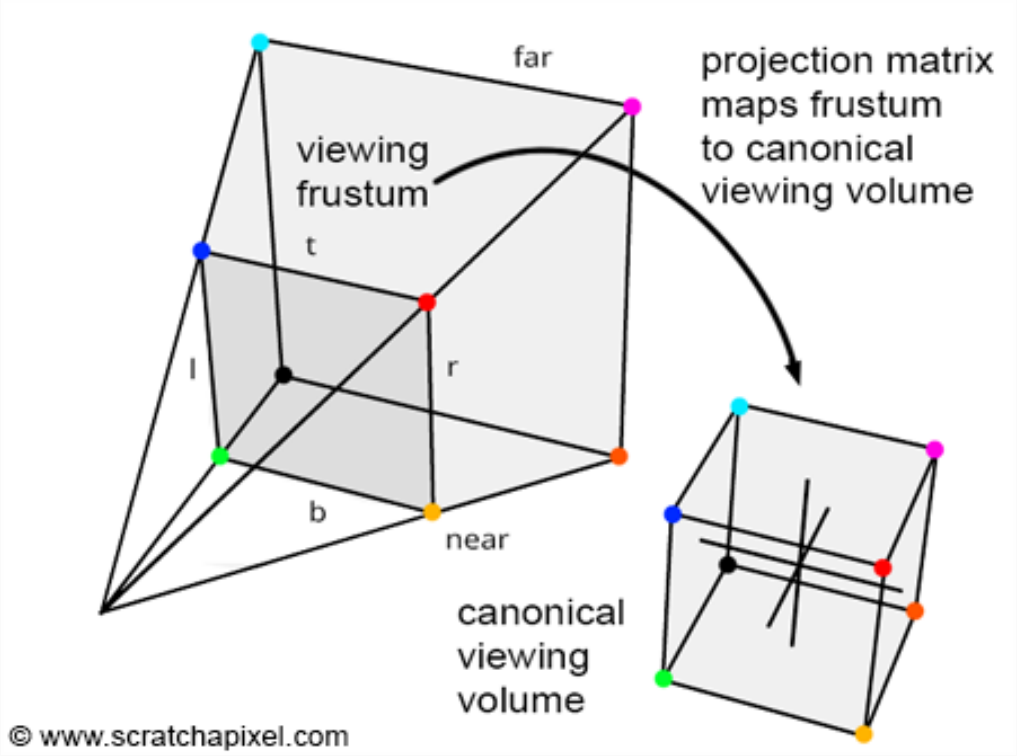
$$[x, y, z, 1]^{\mathrm{T}}] \rightarrow [x, y, z, \frac{z}{f}]^{\mathrm{T}}$$

2. Clip x and y plane of view frustrum
3. Map homogenous coordinate to screen space by homgenising coordinate

$$[x_{clipped}, y_{clipped}, z, \frac{z}{f}]^{\mathrm{T}} \rightarrow [x_{clipped}\frac{f}{z}, y_{clipped}\frac{f}{z}, z\frac{f}{z}, \frac{z}{f}\frac{f}{z}]^{\mathrm{T}} = [u_{clipped}, v_{clipped}, f, 1]^{\mathrm{T}}$$

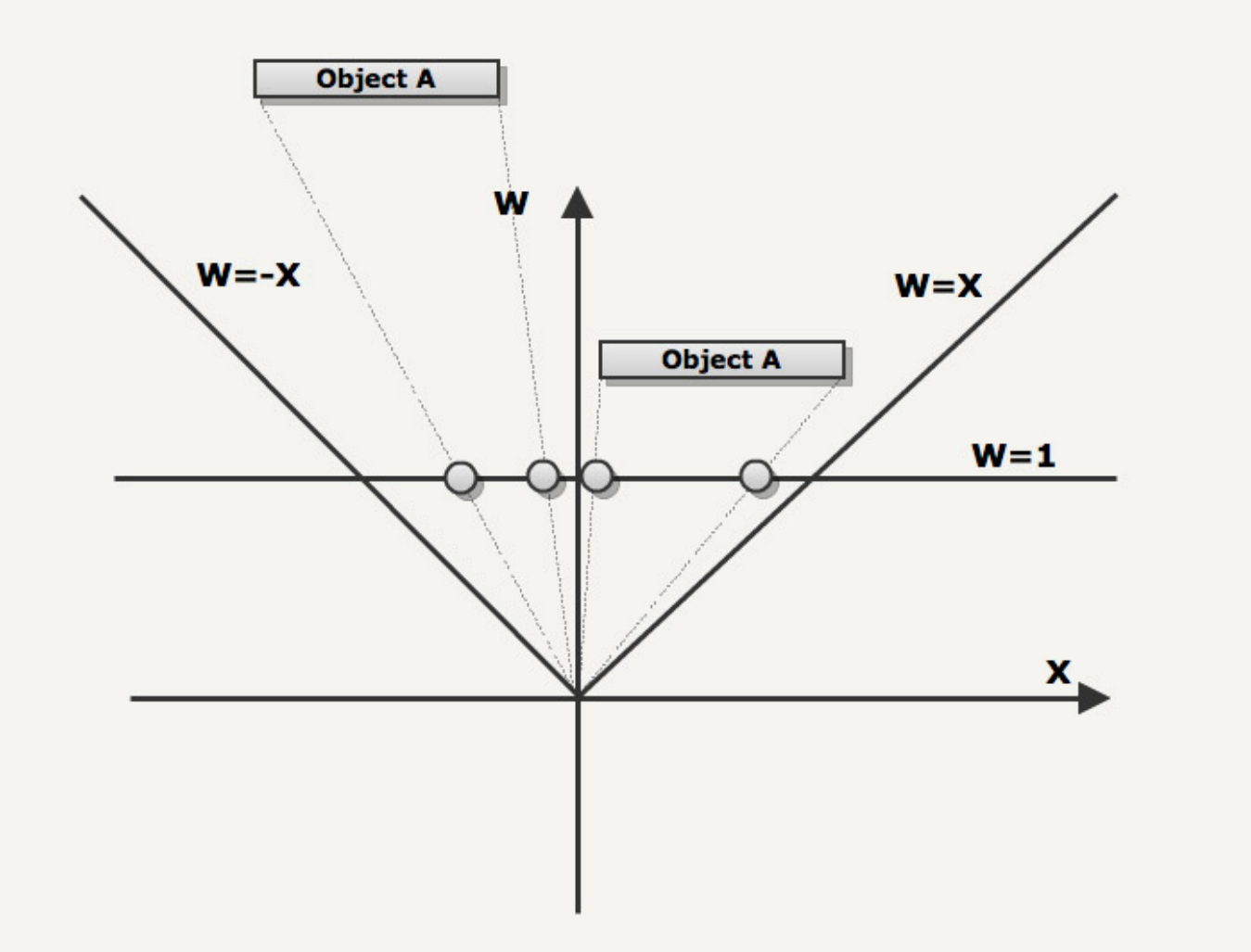
A little bit strange to get ones head around but its easier than one thinks

3D Clipping[3]



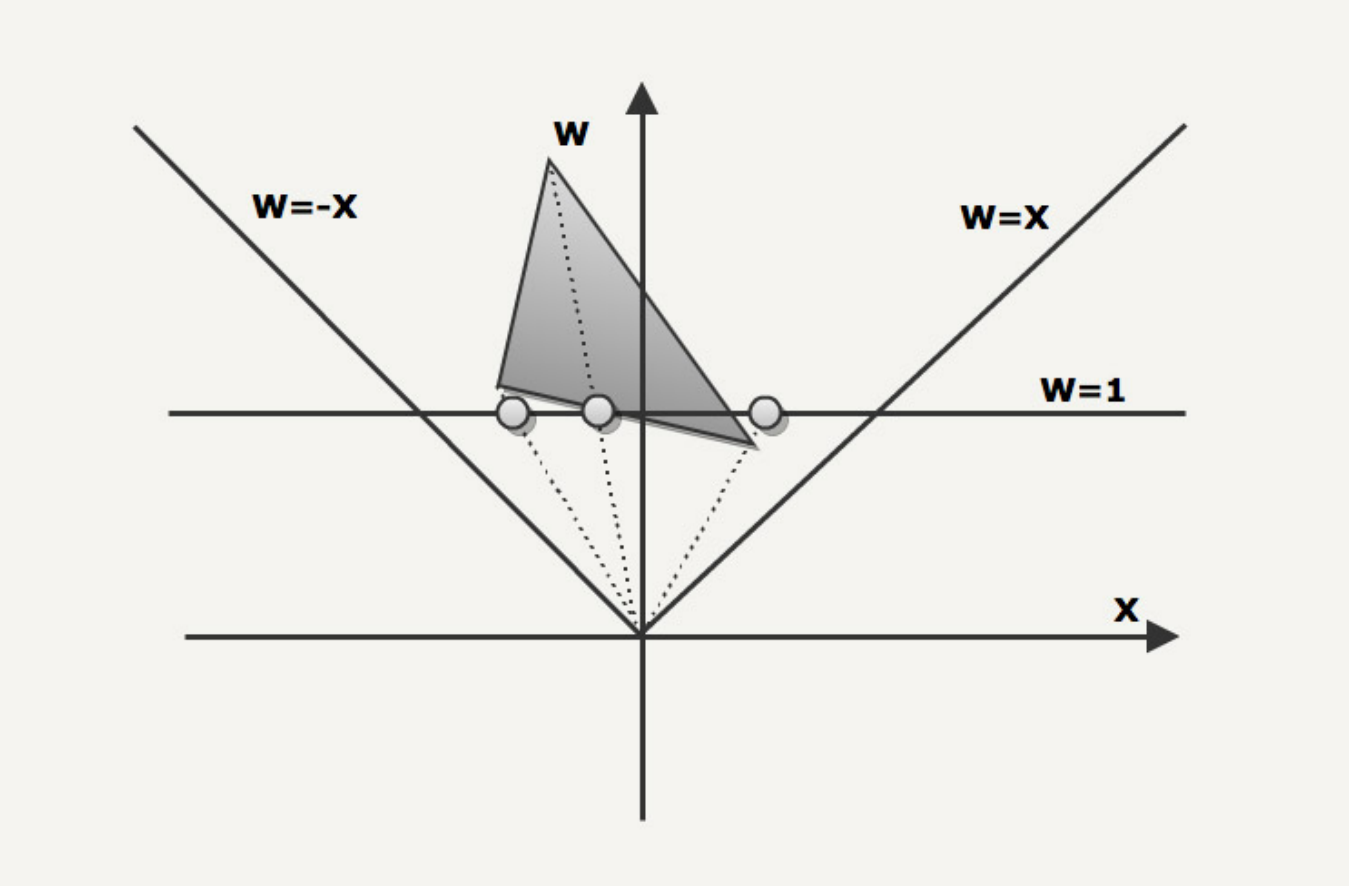
[3] http://fabiensanglard.net/polygon_codec/

3D Clipping[3]



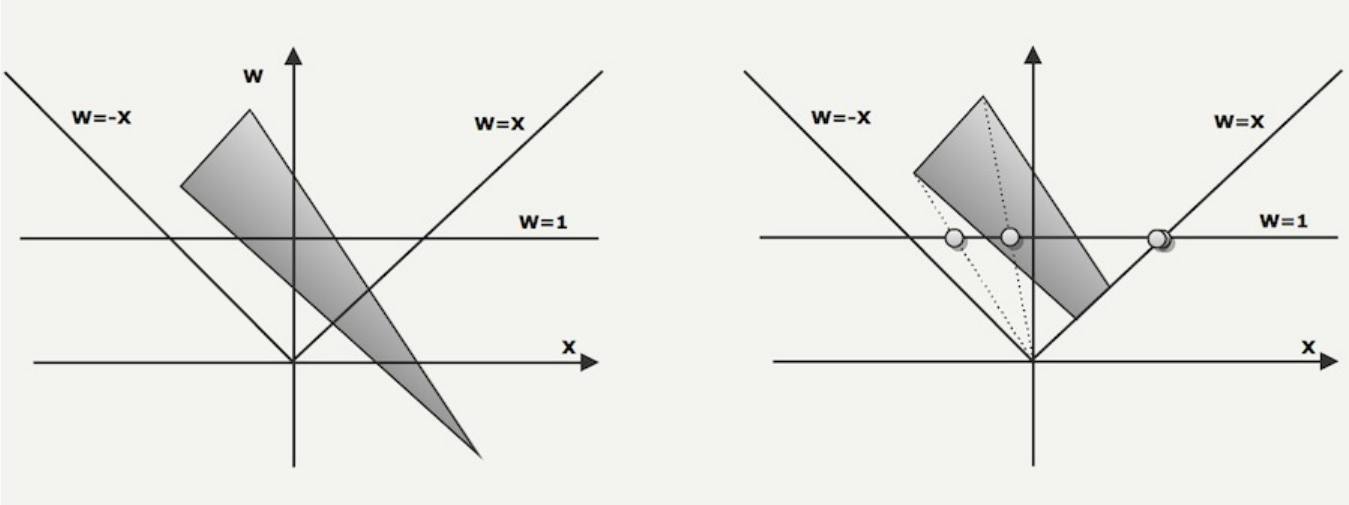
[3] http://fabiensanglard.net/polygon_codec/

3D Clipping[3]



[3] http://fabiensanglard.net/polygon_codec/

3D Clipping[3]



[3] http://fabiensanglard.net/polygon_codec/

Culling

Back-face Remove polygon facing away from camera

- simple, dot product of face normal and view direction positive means not visible
- can we speed up things by clustering normals to remove several directly?

Depth we've already seen this using a z-buffer

Frustrum Remove objects that does not need to be considered for clipping

- construct bounding boxes and compare
- axis aligned or not?

How can we reject groups of objects quickly?

Misc Rendering

Buffers

Frame stores pixels, end product

Depth depth of each pixel

- oclussion culling
- fog
- depth of field, etc.

Stencil discrete buffer per pixel

- mask what to draw in framebuffer
- usually one byte per pixel

Accumulation accumulate frame information

- combine several images

Stencil Buffer

- defines a mask usually `8-bit` which effects if pixel is rendered
- stencil test (same as depth test)
  - `sfail, dpfail, dppass`
  - `GL_KEEP, GL_ZERO, GL_REPLACE, GL_INCR, GL_INCR_WRAP, GL_DECR, GL_DECR_WRAP, GL_INVERT`
- alter stencil based on test result

Stencil Buffer

Reflections

- draw object frame and stencil buffer
- flip object along `z-axis`
- draw reflection area in stencil buffer
- draw flipped object with stencil test such that on floor but not on object

Stencil Shadows[4]

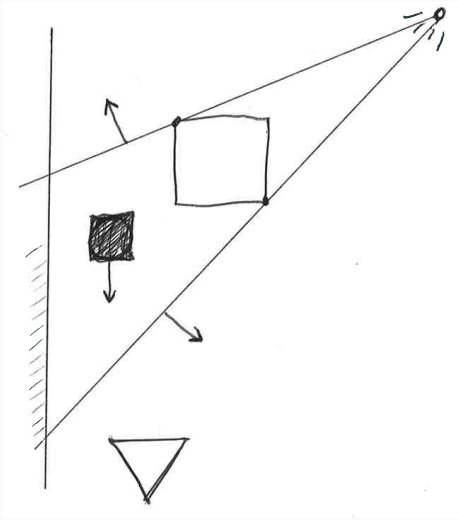
[4] http://archive.gamedev.net/archive/columns/hardcore/shadowvolume/page2.html

Stencil Shadows[4]

1. Draw world in ambient light and update depth buffer (as if the whole world was in shadow)
2. clear stencil buffer
3. Compute shadow volume for each object
4. Draw in stencil buffer for each polygon in shadow volume based on depth test

   fail stencil+$\mathrm{sign}(\mathbf{v}^{\mathrm{T}}\mathbf{n})$

   pass *do nothing*
5. draw specular and diffuse light for non-zero pixels in stencil buffer

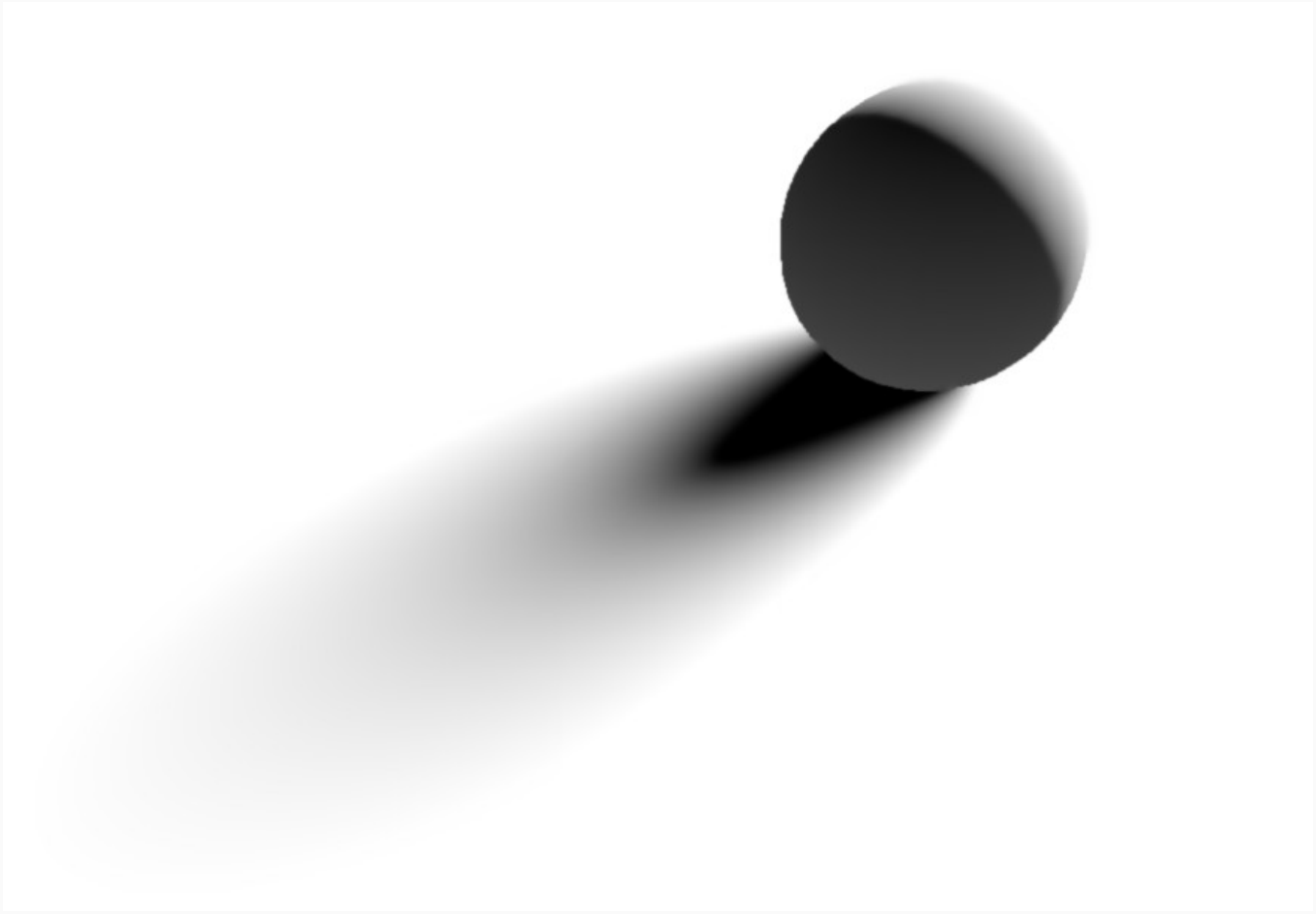


[4] `http://archive.gamedev.net/archive/columns/hardcore/shadowvolume/page2.html`

Accumulation Buffer

- Same size as frame buffer `RGBA`
- Used to do *image processing* on screen
- Always writes every pixel in the buffer
- Very optimised writes (`Blits`) because we are writing a whole chunk of data at the same time
- `void glAccum(GLenum op, float value)`
- `OP: ACCUM, LOAD, RETURN, MULT, ADD`

Combinations



Combinations

Combinations

Multi-pass Rendering

Vertex shader Compute information at vertex

- will interpolate this out across triangle
- linear interpolation (Gouraud)
- if you want something else then you would break up geometry (geometry shader)

Pixel shader Compute information at pixel

- take "any" input **AND** interpolated vertex attribute
- sometimes necessary or useful to break up computation in several stages

Run shader in several passes across polygon. We did just that with first Z and then vertex attributes.

Summary

End of Part II

- *Computer* Graphics
  - generate graphics in a manner suitable for commputers
- Sparse (per vertex) computations of light
- Interpolate vertex attributes across pixels (fragments)

End of Part II

- *Computer* Graphics
  - generate graphics in a manner suitable for commputers
- Sparse (per vertex) computations of light
- Interpolate vertex attributes across pixels (fragments)
- We have seen at least one example of each part of the pipeline but there are many many more versions. Choose algorithm based on the hardware that you program.

Rasterisation

- This is how the internals of game engines work
- This is what your GPU does for you (a lot of it at least)
- Now you know how this works
  - my hope have been that this should allow you to make more efficient use of modern APIs
  - understand how and what you can tweak
  - understand how you can exploit things to your benefit
- Fixed Rendering Pipeline is no more

Course so far

Raytracer how is an image generated

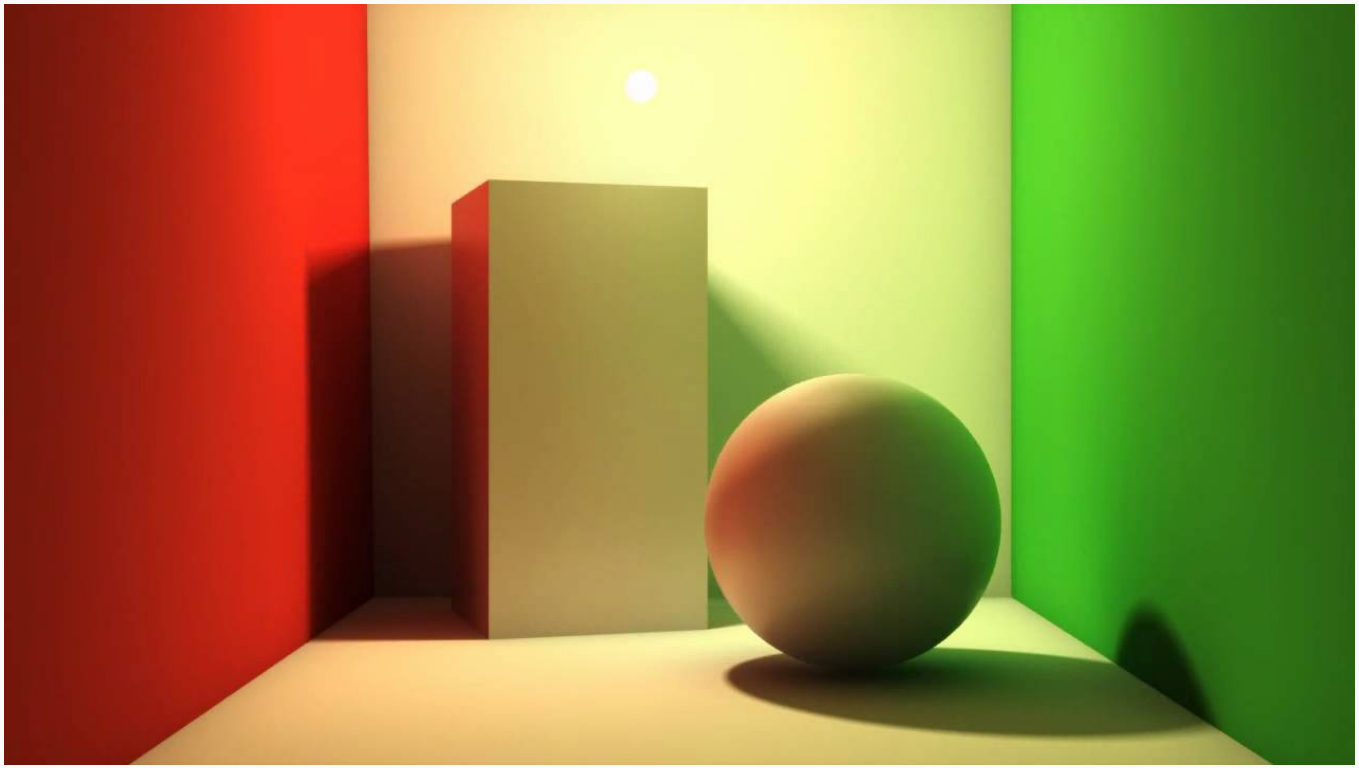
- images are actually quite simple
- only thing holding back realism is computation time

Raster how does a computer display an image

- in lab you have seen what the basics are
- lectures have gone a few steps further

Hopefully you feel that you understand how the graphics pipeline works and more importantly have ideas thought on how to make each step in the lab better.

Part II



$$L(x \to \Theta) = L_e(x \to \Theta) + \int_{\Omega_x} f_r(x, \Psi \to \Theta) L(x \leftarrow \Psi) \cos(\mathbf{n}_x, \Psi) \mathrm{d}\omega_\Psi$$

Global Illumination

- What happens when light hits a surface
  - Light comes in reflects and refracts
  - Light emits from point
- Ammount of "light" constant in a closed environment
- Solve for this steady-state
- Approximate integral

Next Time

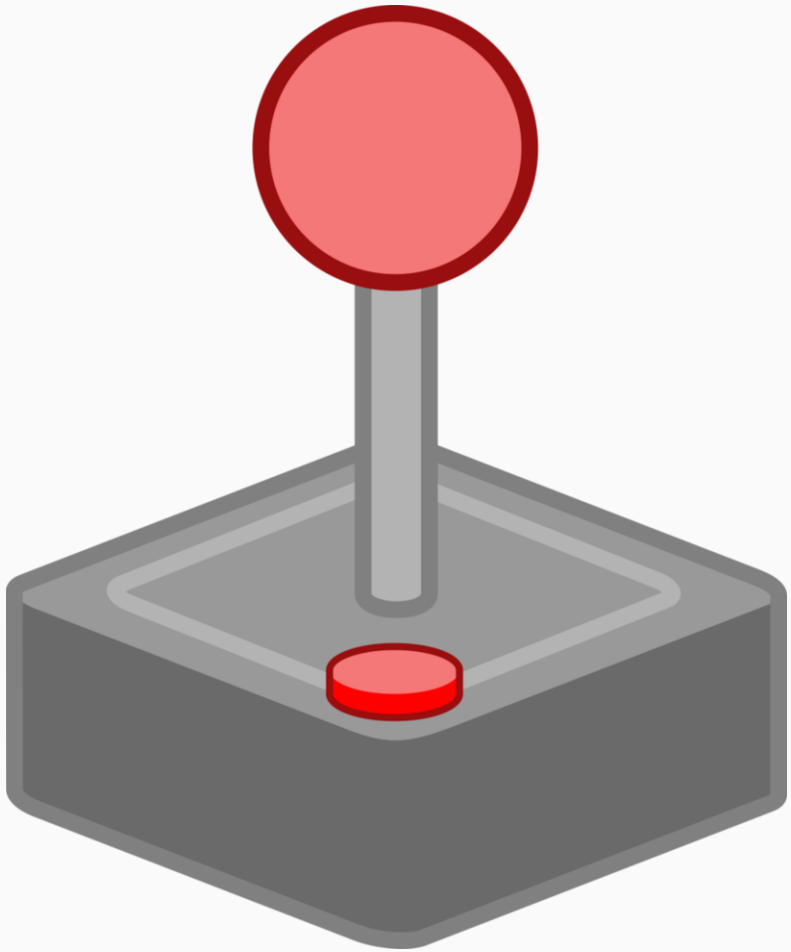
Lecture Global Illumination

- Introduce concepts
- Formulate problem

Lab Rasterisation

- Try to finish 50% mark of both courseworks this week if you aim for extensions

Homework



END

eof