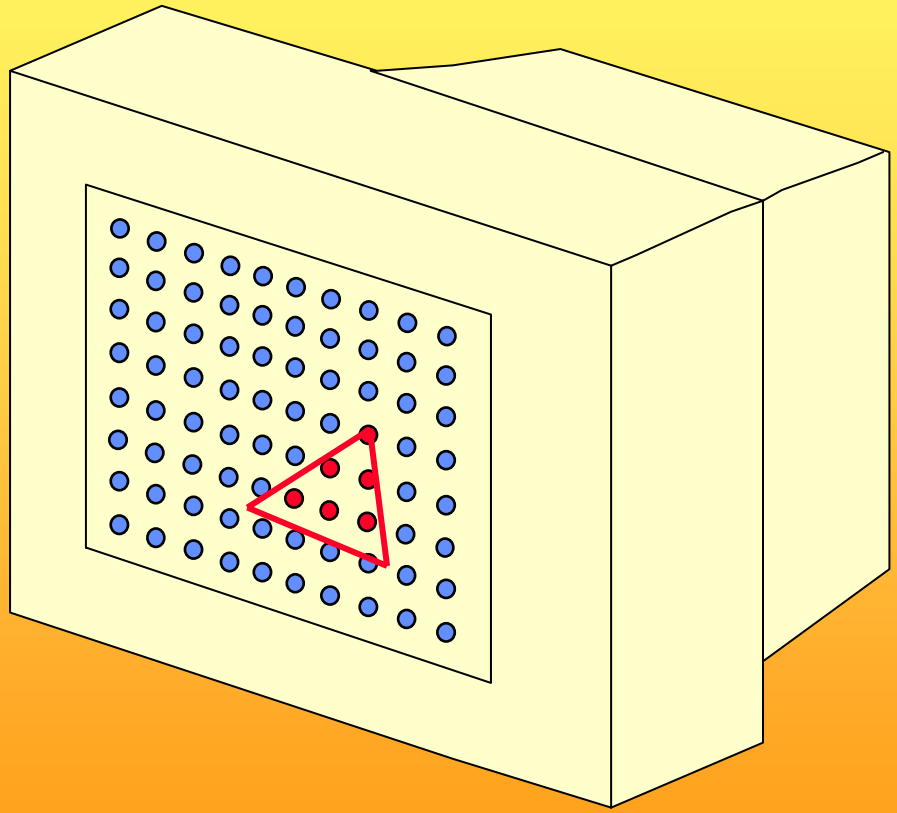
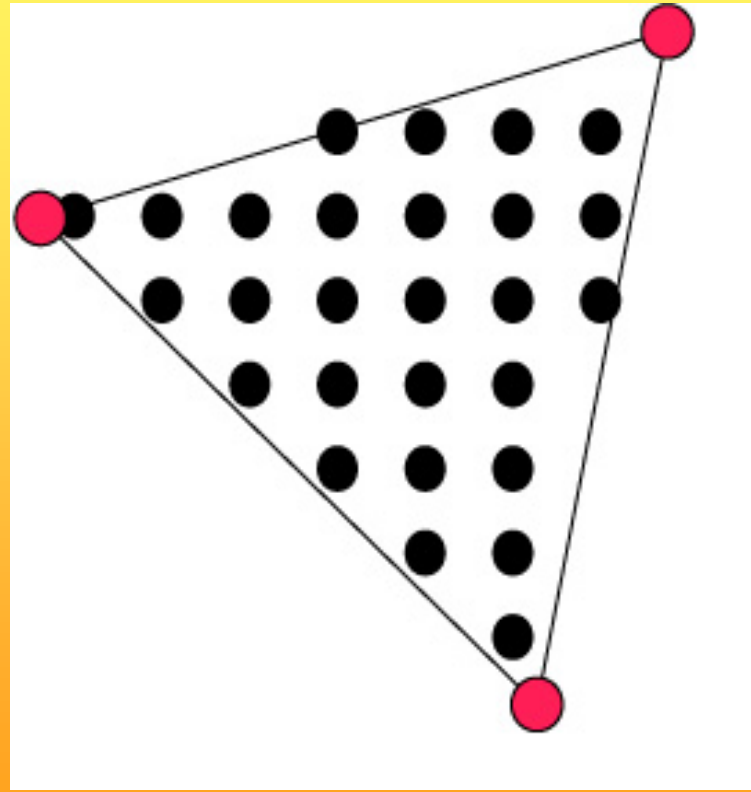
Overview of the Pipeline and OpenGL

Allmänt

- Labkonto
- Inpasseringskort
  - aktiveras automatiskt till alla som är kursregistrerade som har CTH/GU-inpasseringskort
- Labgrupper – helst 2 o 2
  - Skriv upp er på lista här
- Kursregistreringslista

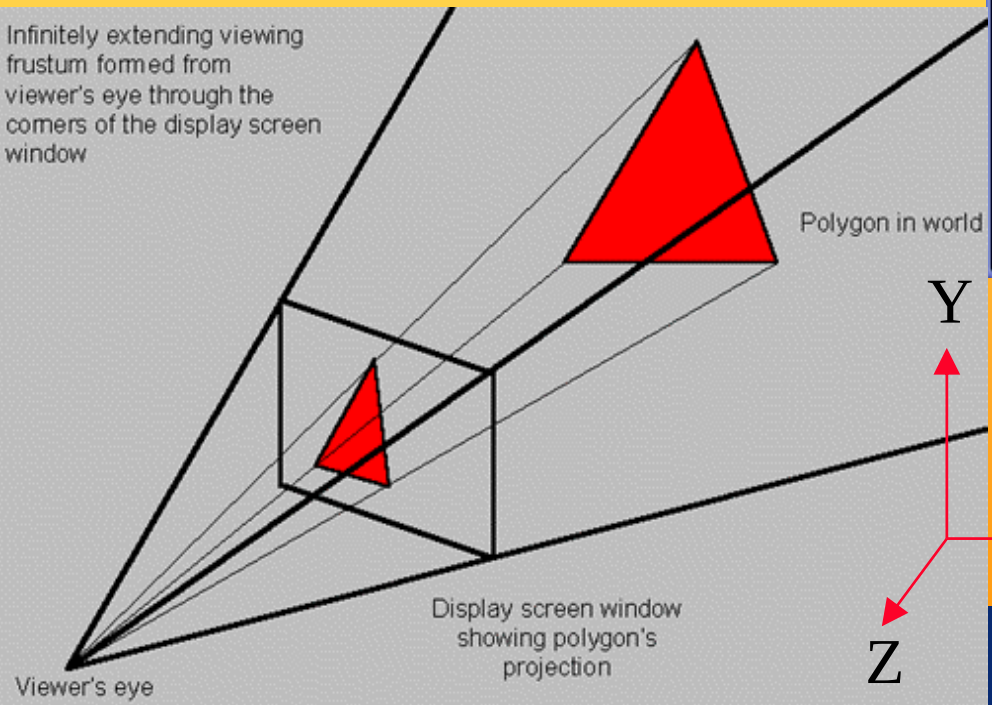
Skärmen består av massa pixlar

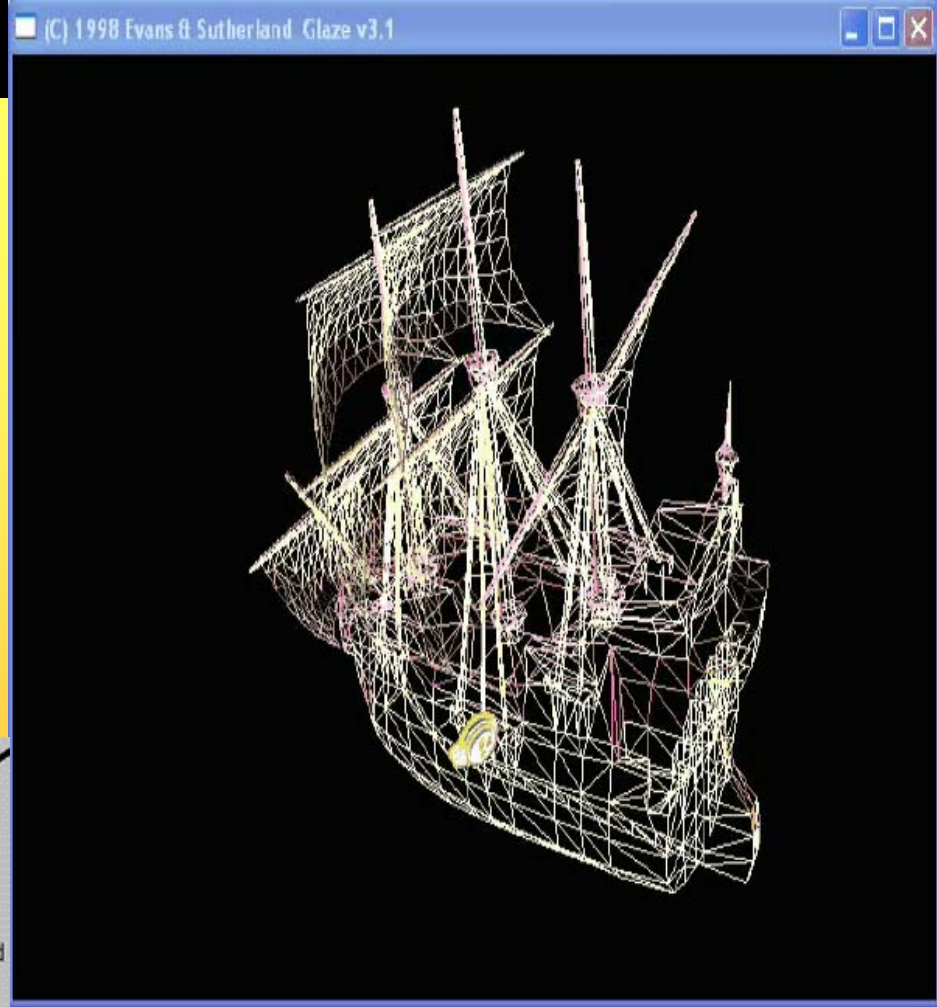




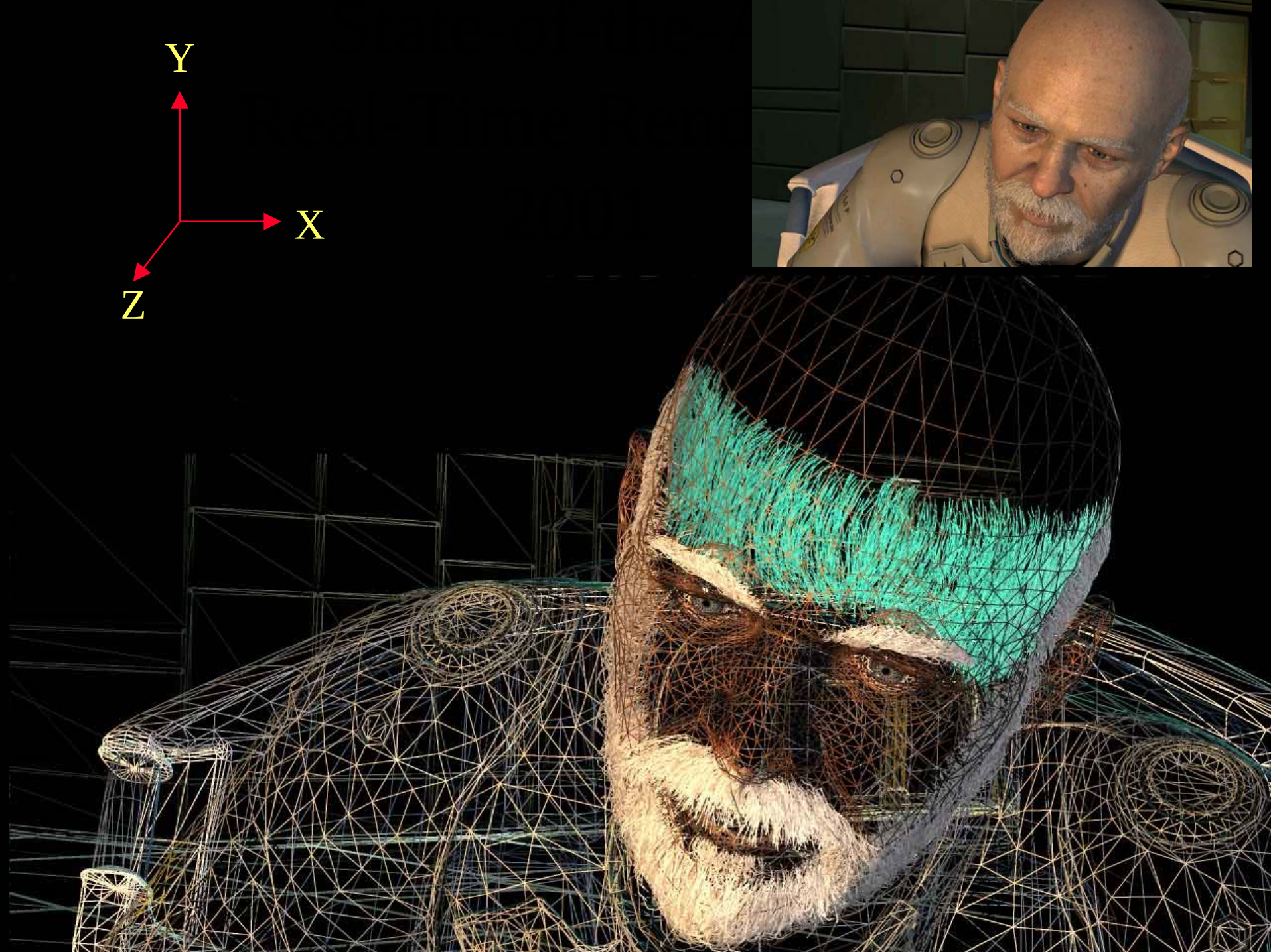
3D-Rendering

- Objects are often made of triangles
- x,y,z- coordinate for each vertex



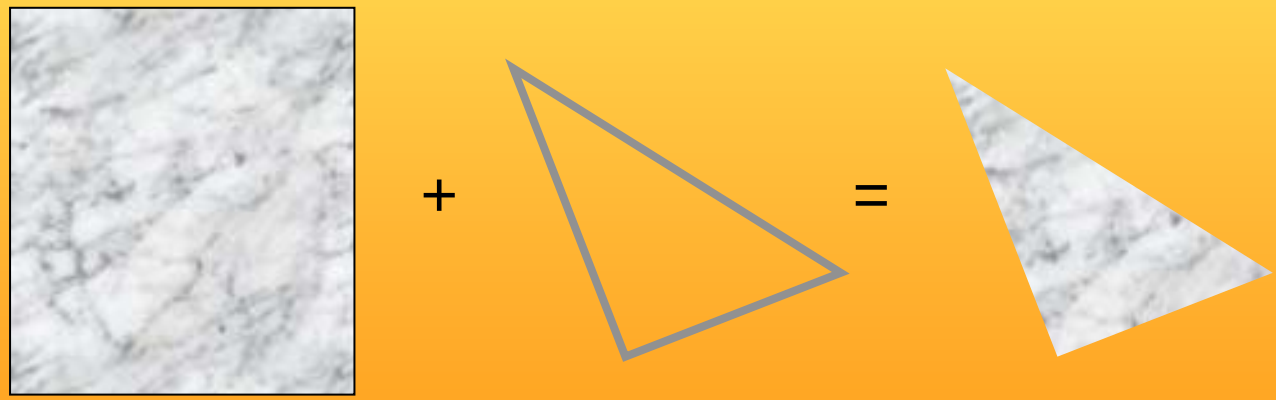


State-of-the-Art Real-Time Rendering 2001


Y
X
Z

Textures

- One application of texturing is to ”glue” images onto geometrical object

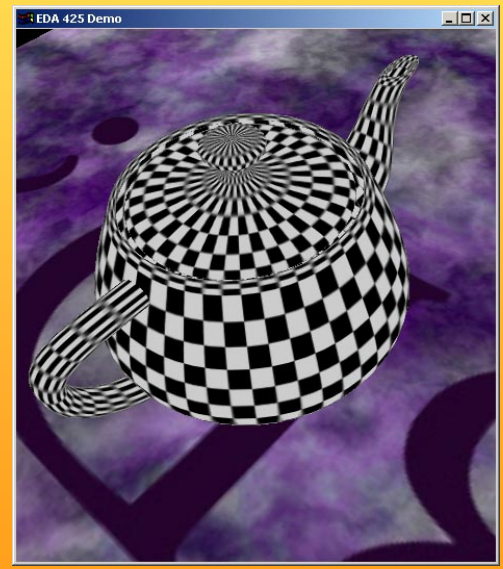


Texturing: Glue images onto geometrical objects

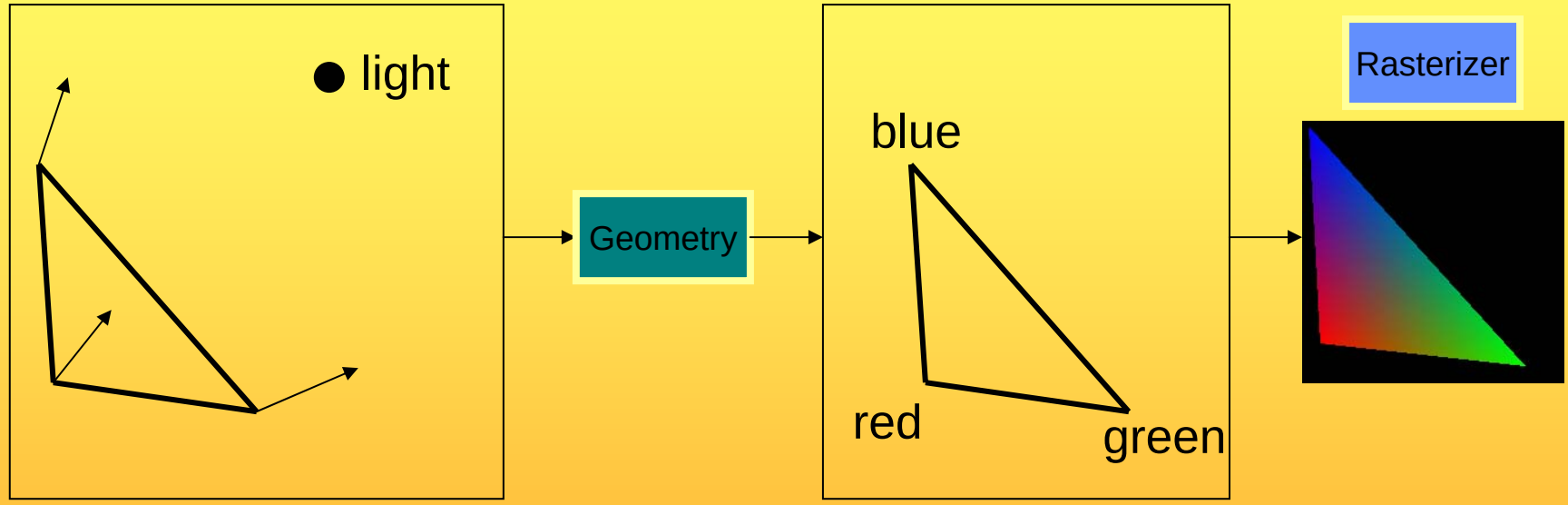
- Purpose: more realism, and this is a cheap way to do it

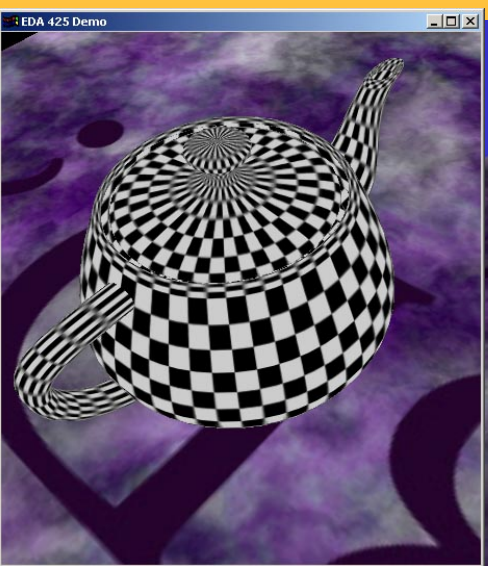
+

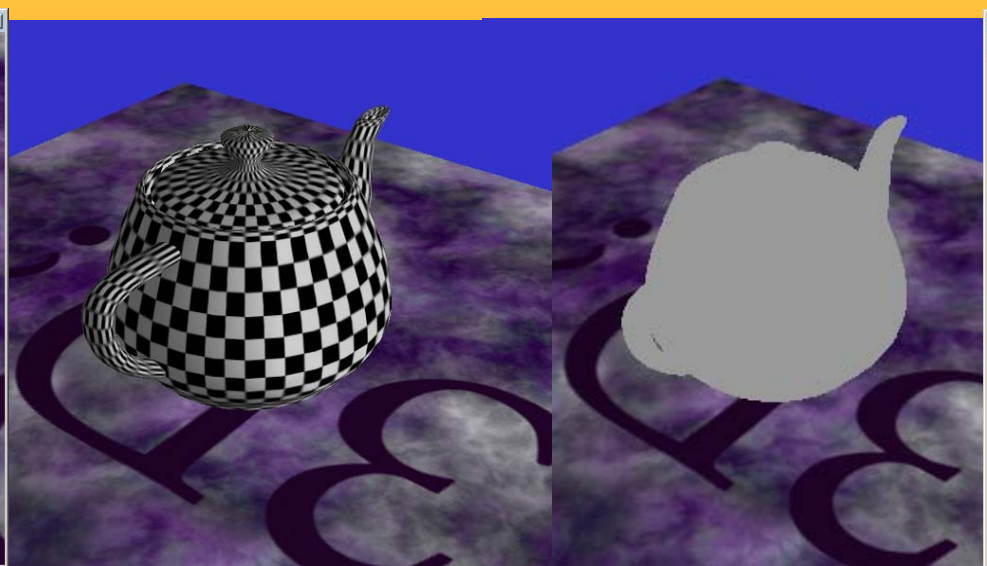
=



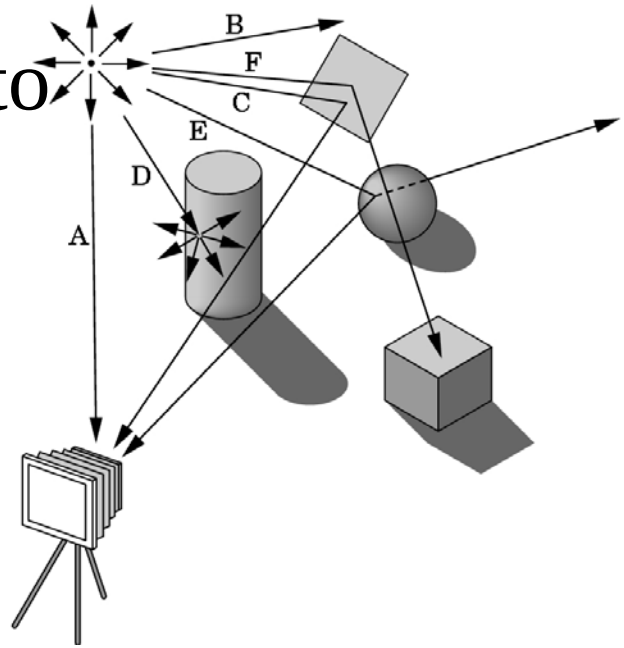
Ljusberäkning per triangelhörn







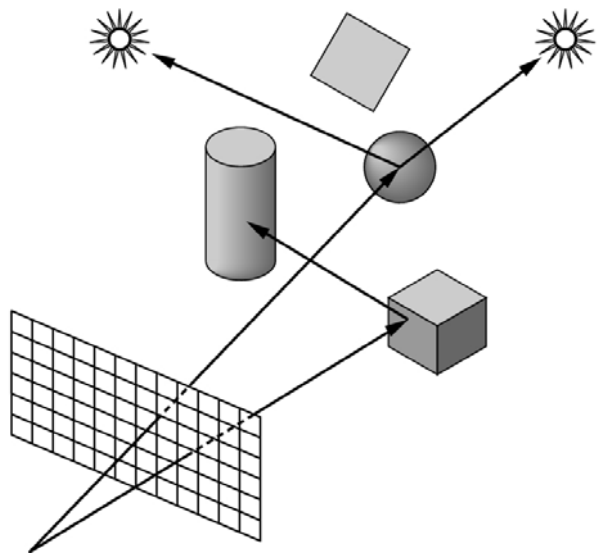
Ray Tracing and Geometric Optics



One way to form an image is to follow rays of light from a point source finding which rays enter the lens of the camera. However, each ray of light may have multiple interactions with objects before being absorbed or going to infinity.

Physical Approaches

- **Ray tracing**: follow rays of light from center of projection until they either are absorbed by objects or go off to infinity
  - Can handle global effects
    - Multiple reflections
    - Translucent objects
  - Slow
  - Must have whole data base available at all times

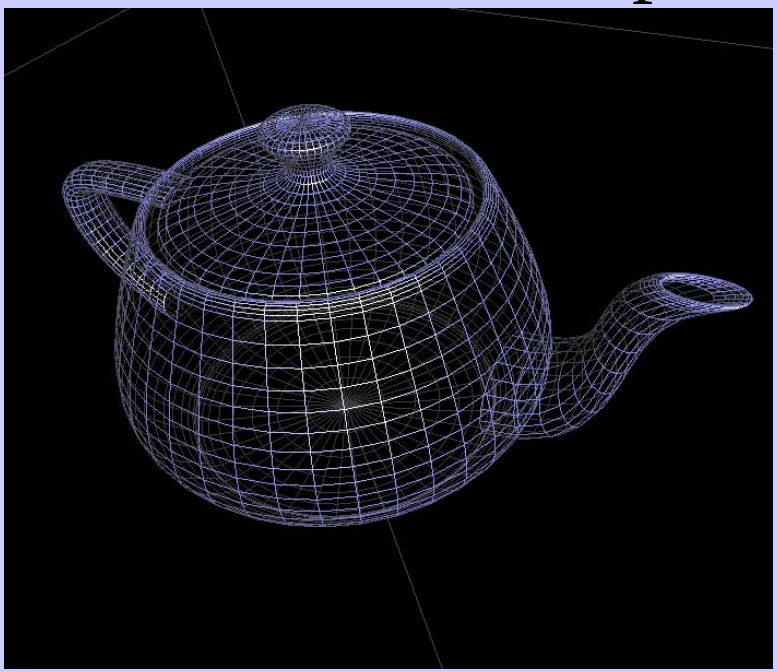


- **Radiosity**: Energy based approach
  - Very slow

The Graphics Rendering Pipeline

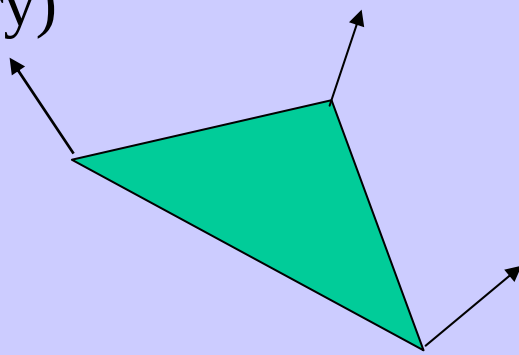
Rendering Primitives

- Use graphics hardware for real time…
- These can render points, lines, triangles.
- A surface is thus an approximation by a number of such primitives.



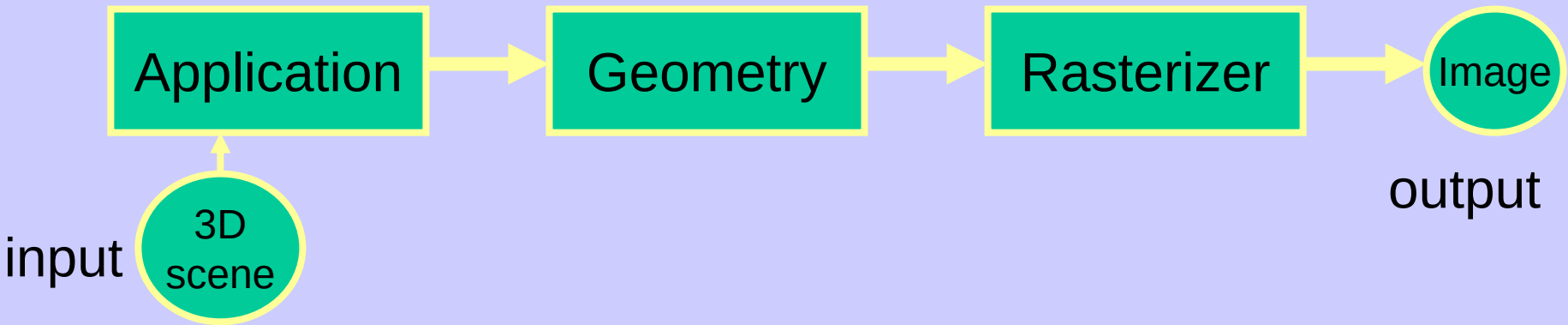
You say that you render a *”3D scene”*, but what is it?

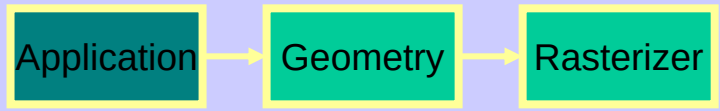
- First, of all to take a picture, it takes a camera – a virtual one.
  - Decides what should end up in the final image
- A 3D scene is:
  - Geometry (triangles, lines, points, and more)
  - Light sources
  - Material properties of geometry
  - Textures (images to glue onto the geometry)
- A triangle consists of 3 vertices
  - A vertex is 3D position, and may include normals and more.



Lecture 1: Real-time Rendering
The Graphics Rendering Pipeline

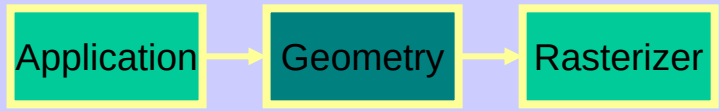
- The pipeline is the ”engine” that creates images from 3D scenes
- Three conceptual stages of the pipeline:
  - Application (executed on the CPU)
  - Geometry
  - Rasterizer





The APPLICATION stage

- Executed on the CPU
  - Means that the programmer decides what happens here
- Examples:
  - Collision detection
  - Speed-up techniques
  - Animation
- Most important task: send rendering primitives (e.g. triangles) to the graphics hardware

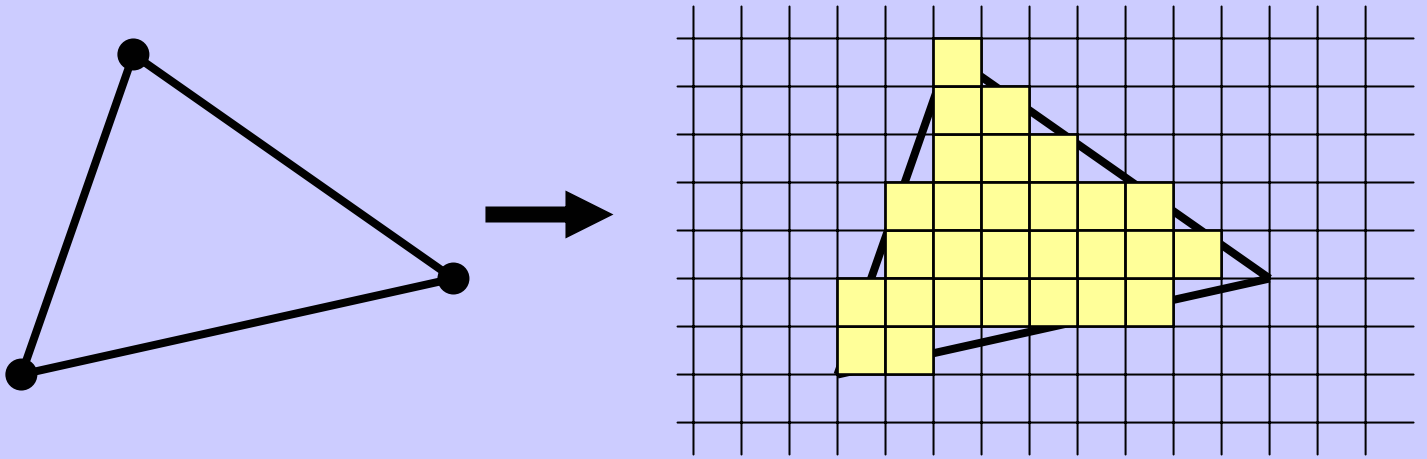


The GEOMETRY stage

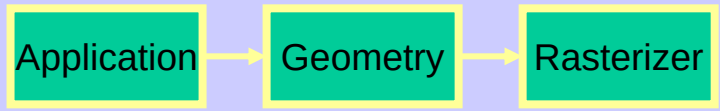
- Task: ”geometrical” operations on the input data (e.g. triangles)
- Allows:
  - Move objects (matrix multiplication)
  - Move the camera (matrix multiplication)
  - Compute lighting at vertices of triangle
  - Project onto screen (3D to 2D)
  - Clipping (avoid triangles outside screen)
  - Map to window

The RASTERIZER stage

- Main task: take output from GEOMETRY and turn into visible pixels on screen



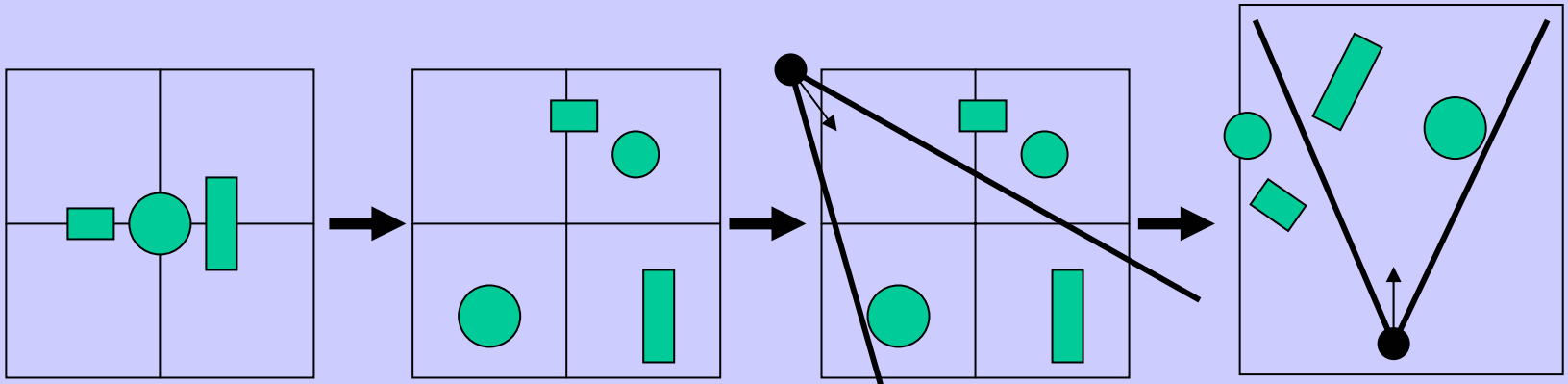
  - Also, add textures and various other per-pixel operations
  - And visibility is resolved here: sorts the primitives in the z-direction

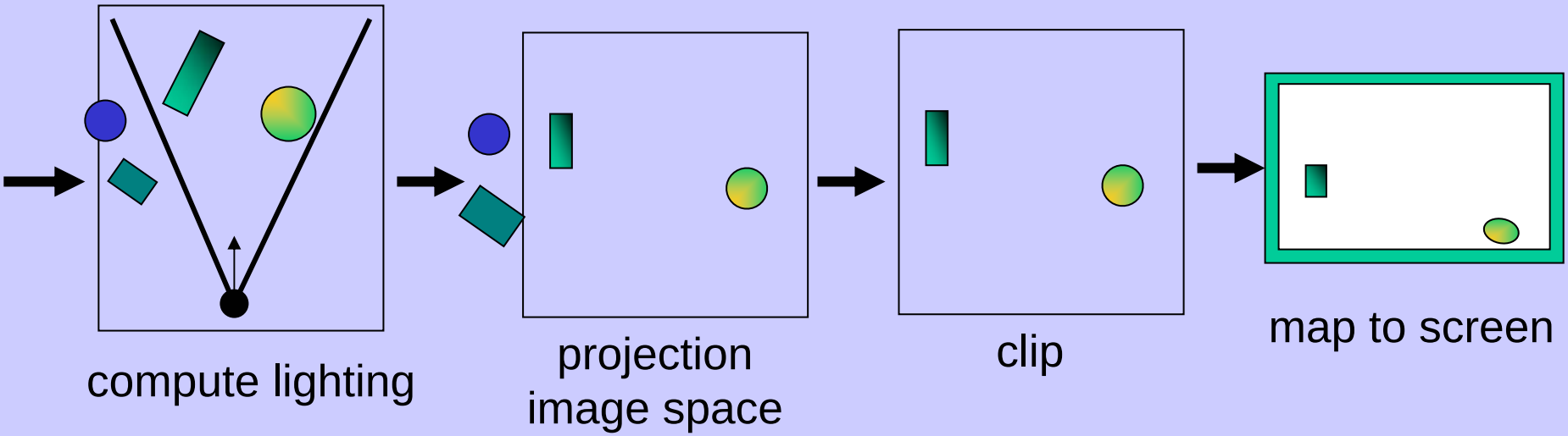


Rewind!
Let’s take a closer look

- The programmer ”sends” down primtives to be rendered through the pipeline (using API calls)
- The geometry stage does per-vertex operations
- The rasterizer stage does per-pixel operations
- Next, scrutinize geometry and rasterizer

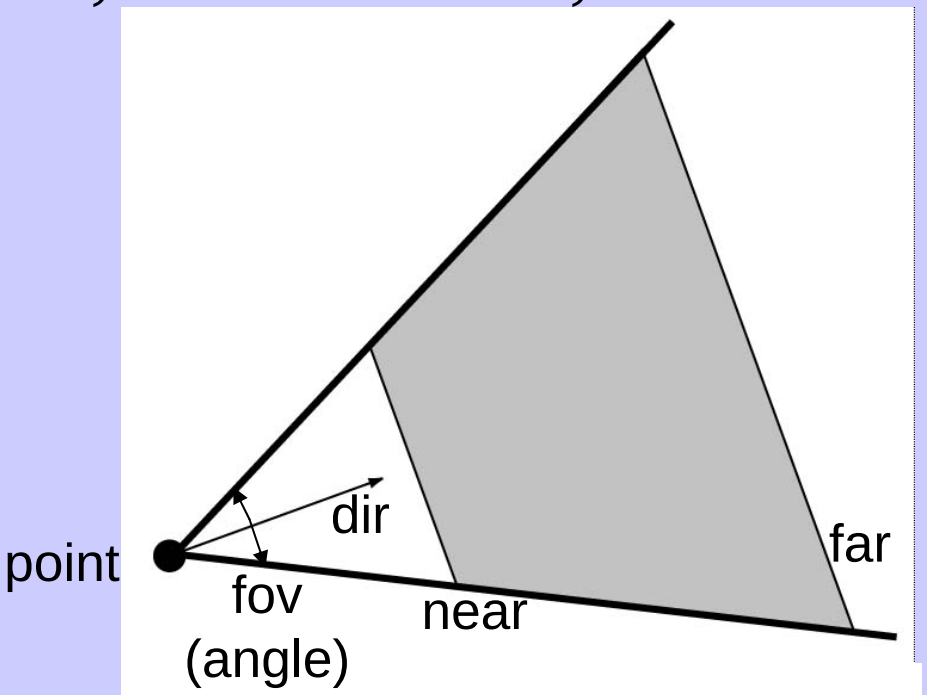
GEOMETRY - Summary





Virtual Camera

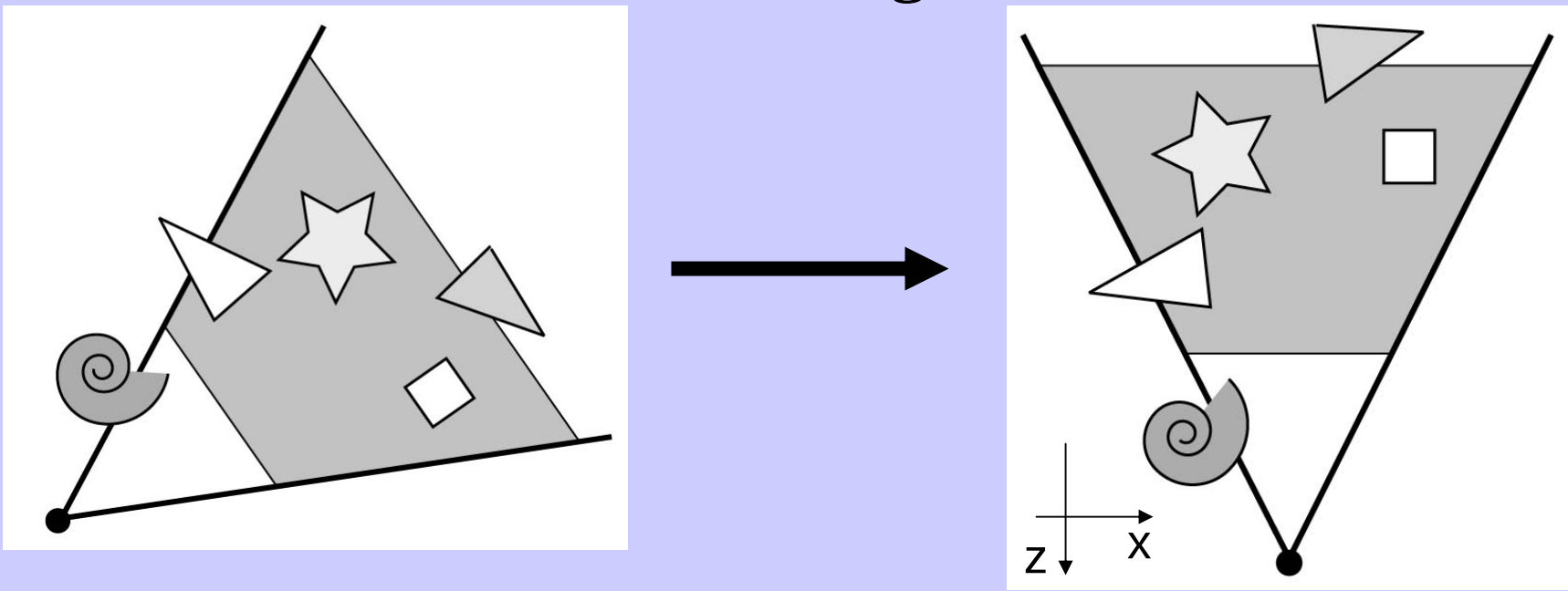
- Defined by position, direction vector, up vector, field of view, near and far plane.



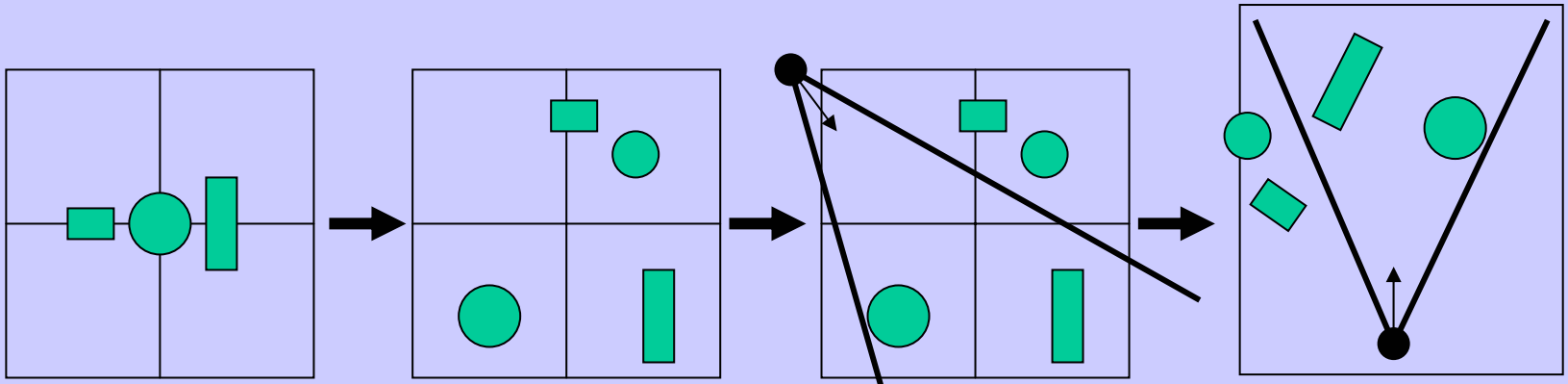
- Create image of geometry inside gray region
- Used by OpenGL, DirectX, ray tracing, etc.

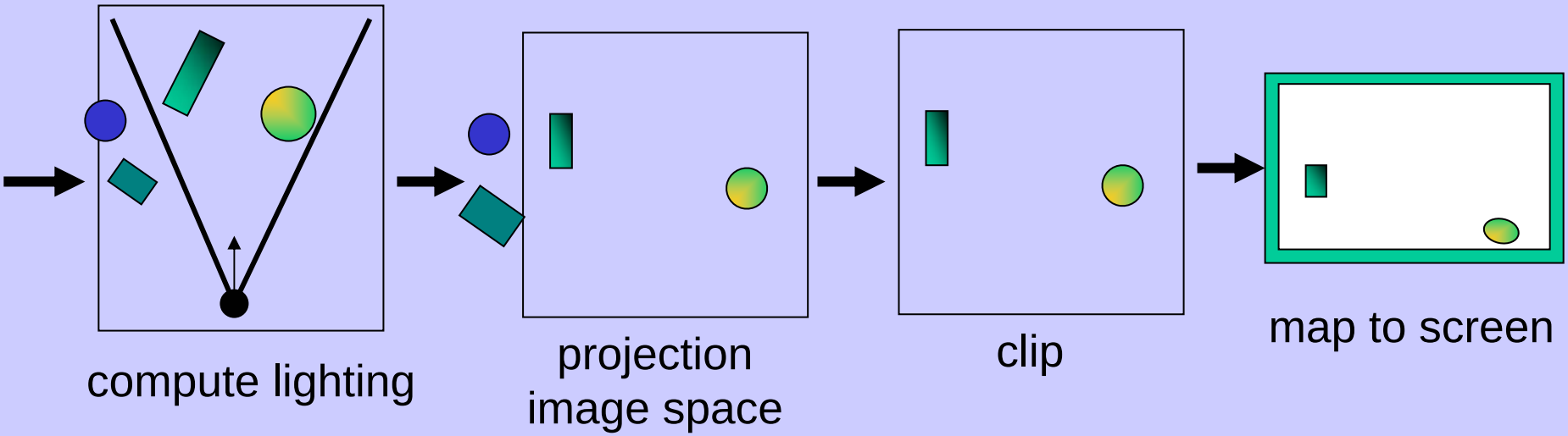
GEOMETRY - The view transform

- You can move the camera in the same manner as objects
- But apply inverse transform to objects, so that camera looks down negative z-axis



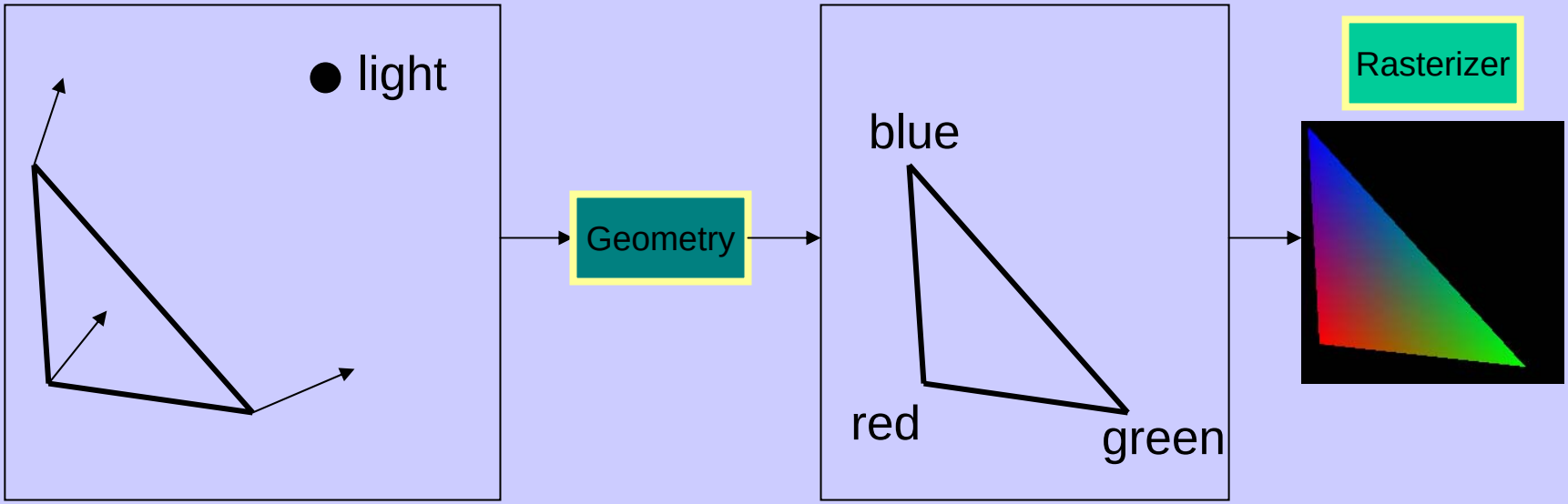
GEOMETRY - Summary





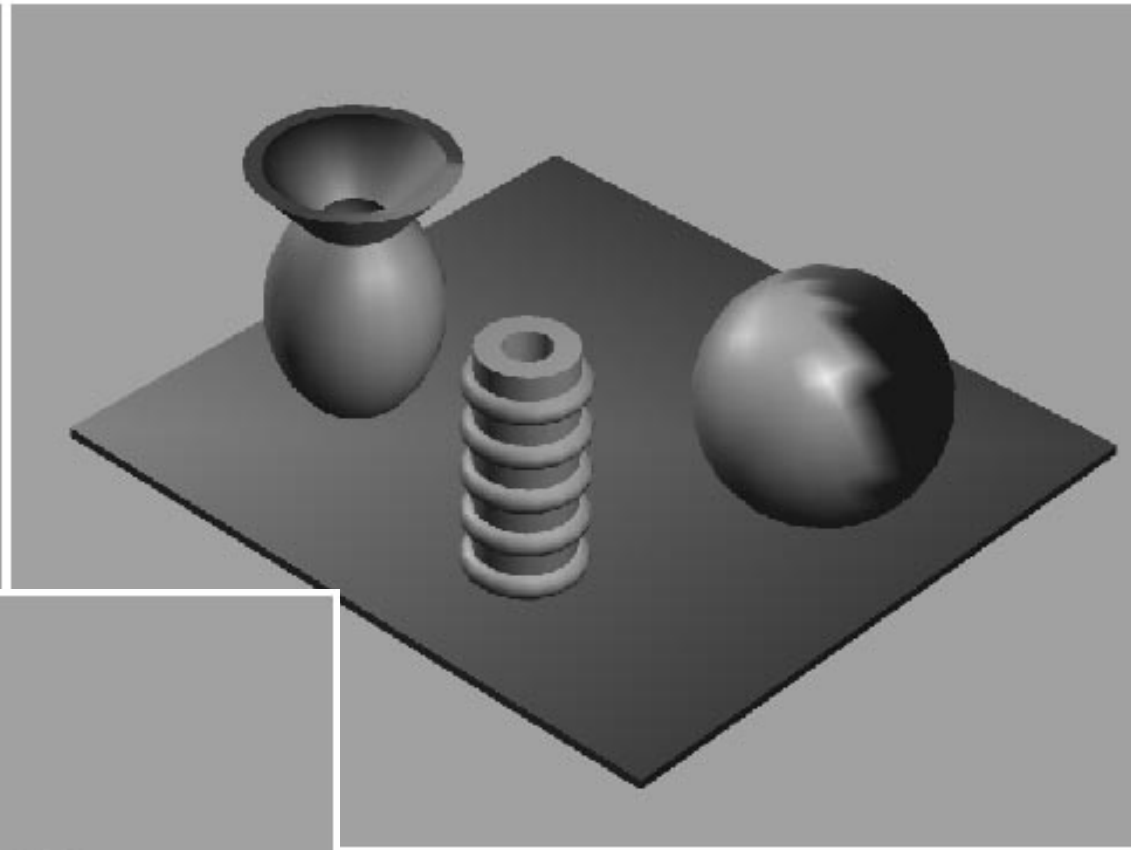
GEOMETRY - Lighting

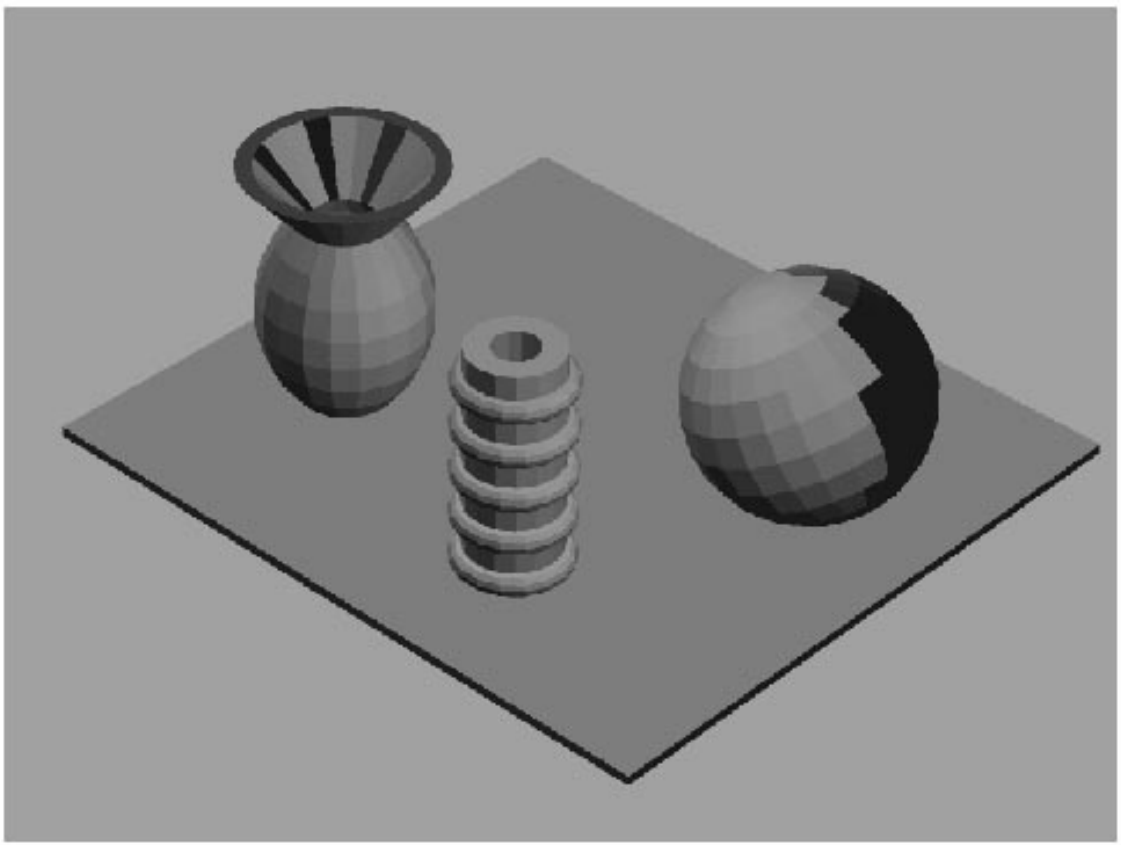
- Compute ”lighting” at vertices



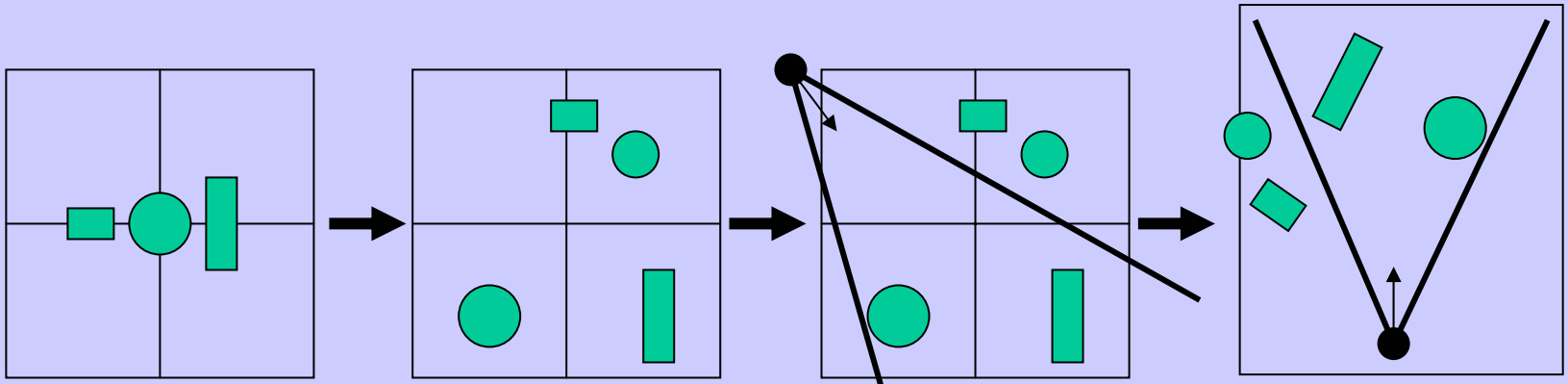
- Try to mimic how light in nature behaves
  - Hard so uses empirical models, hacks, and some real theory
- Much more about this in later lecture

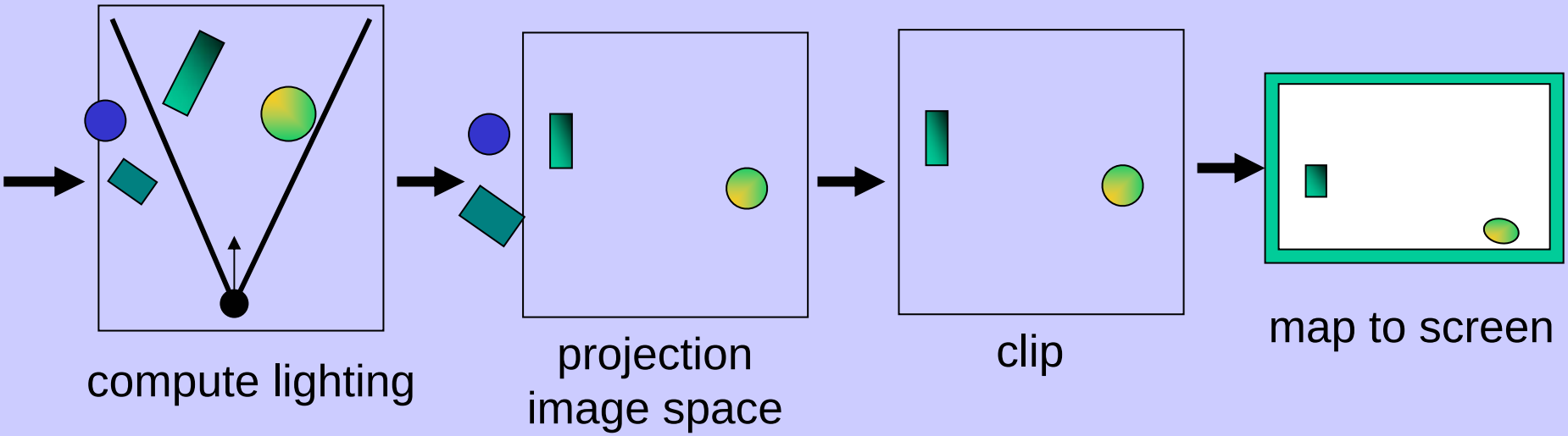
Example





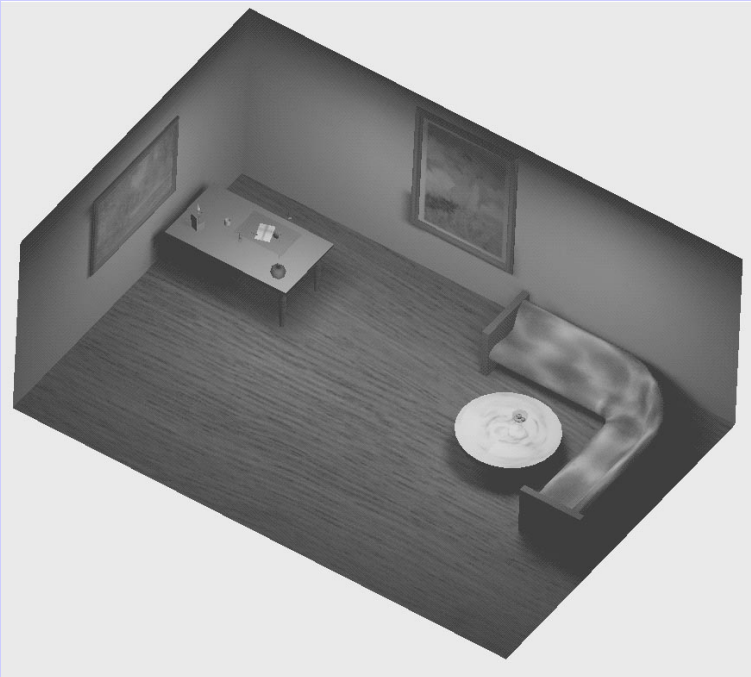
GEOMETRY - Summary

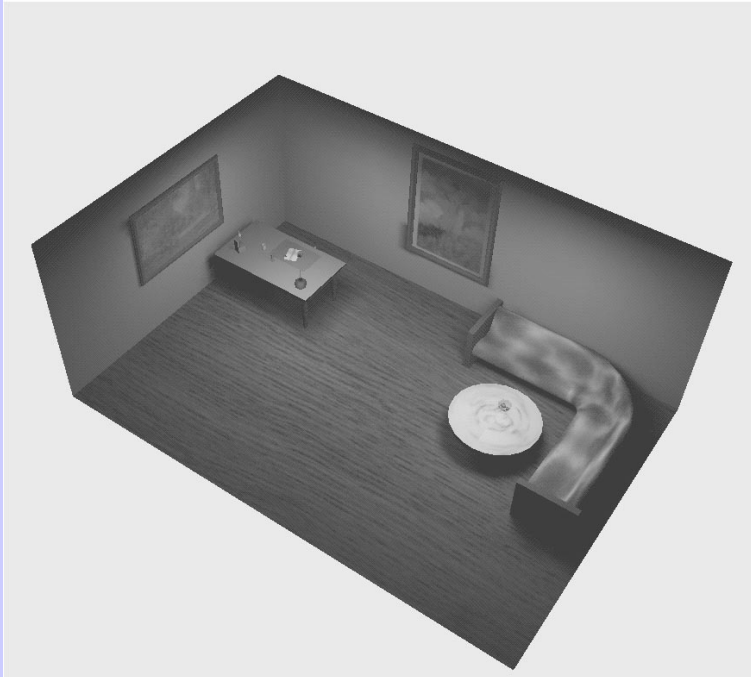




GEOMETRY - Projection

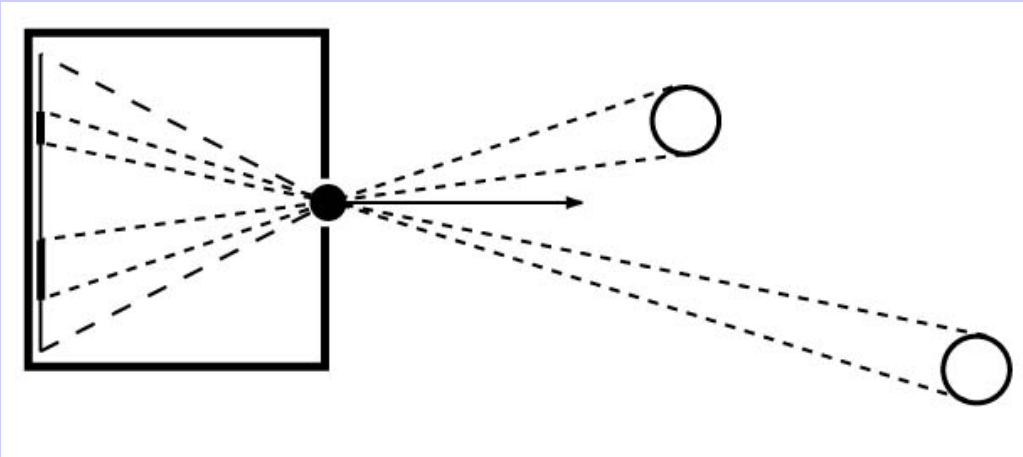
- Two major ways to do it
  - Orthogonal (useful in few applications)
  - Perspective (most often used)
    - Mimics how humans perceive the world, i.e., objects' apparent size decreases with distance

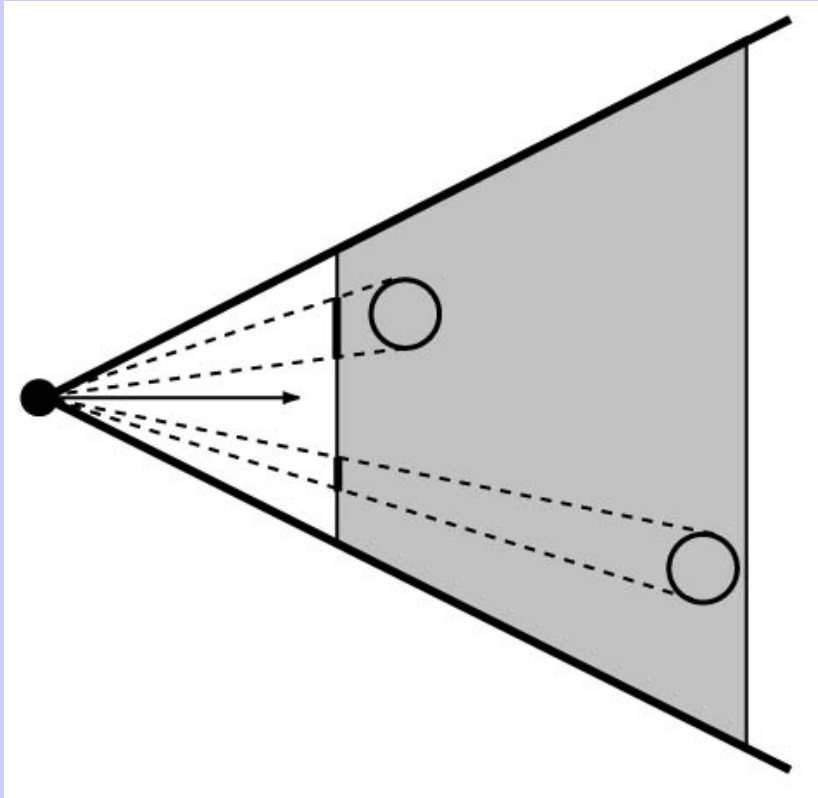




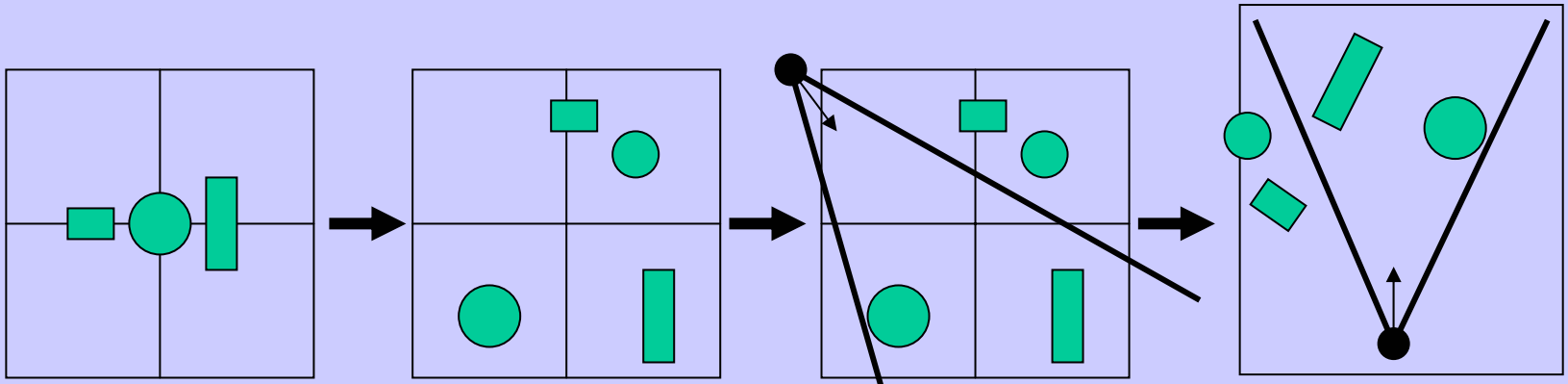
GEOMETRY - Projection

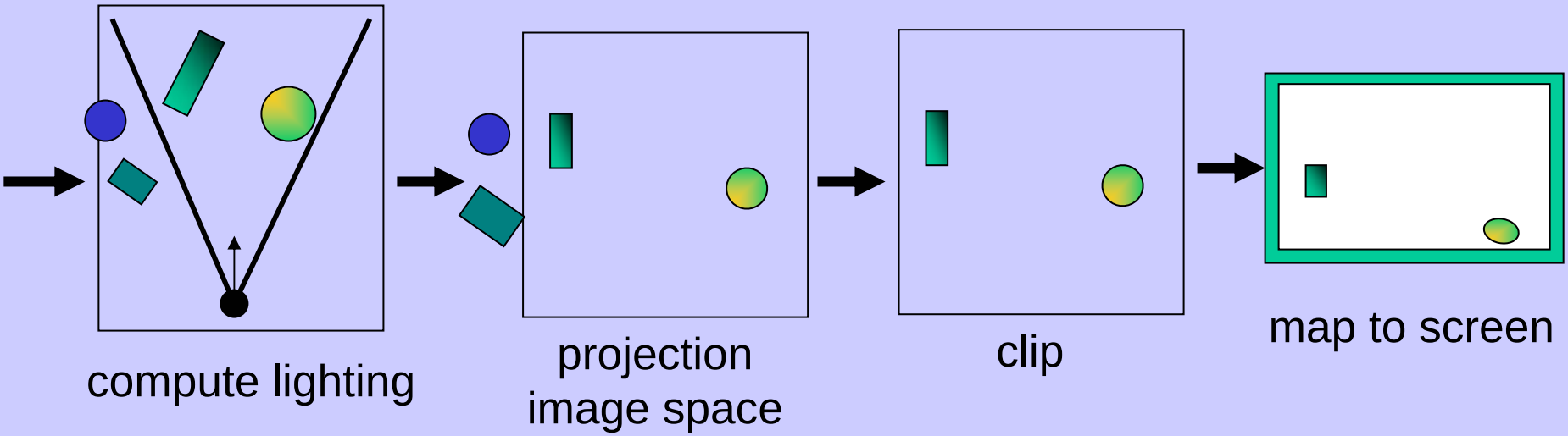
- Also done with a matrix multiplication!
- Pinhole camera (left), analog used in CG (right)



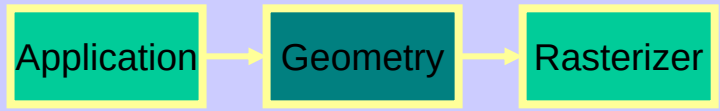


GEOMETRY - Summary



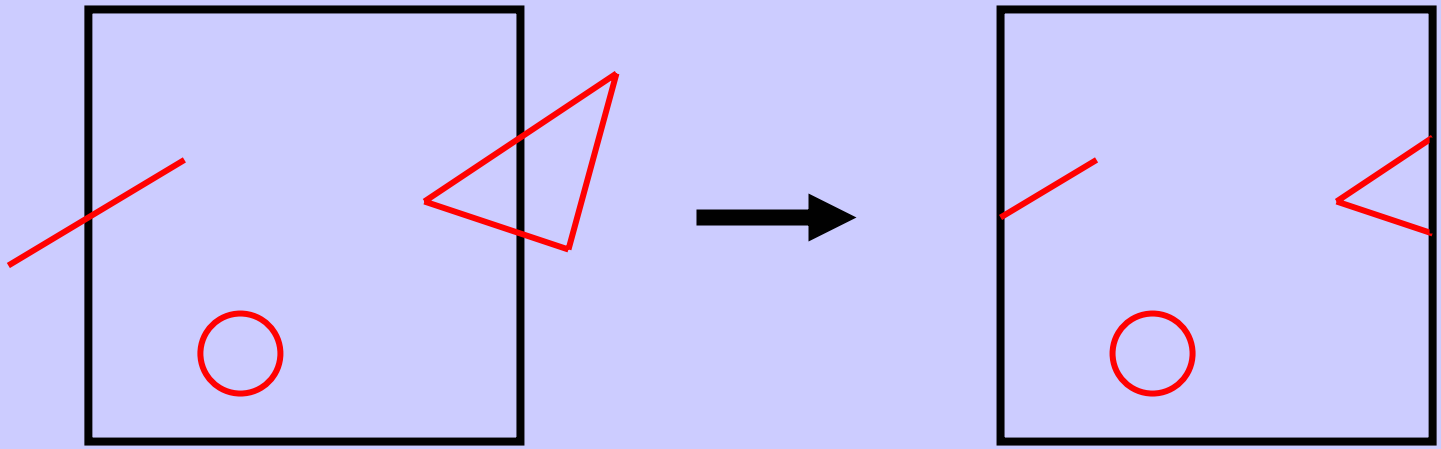


GEOMETRY



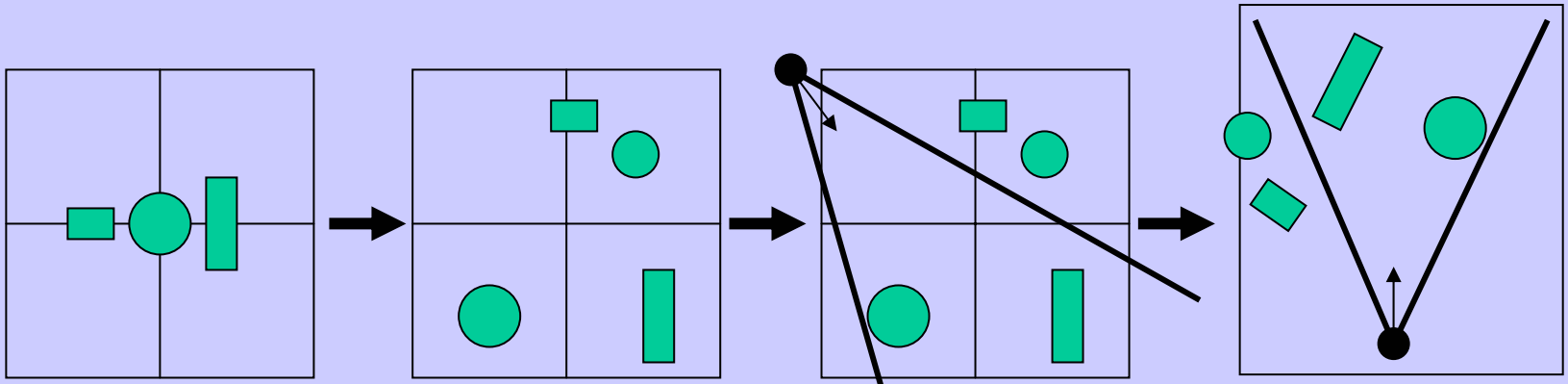
Clipping and Screen Mapping

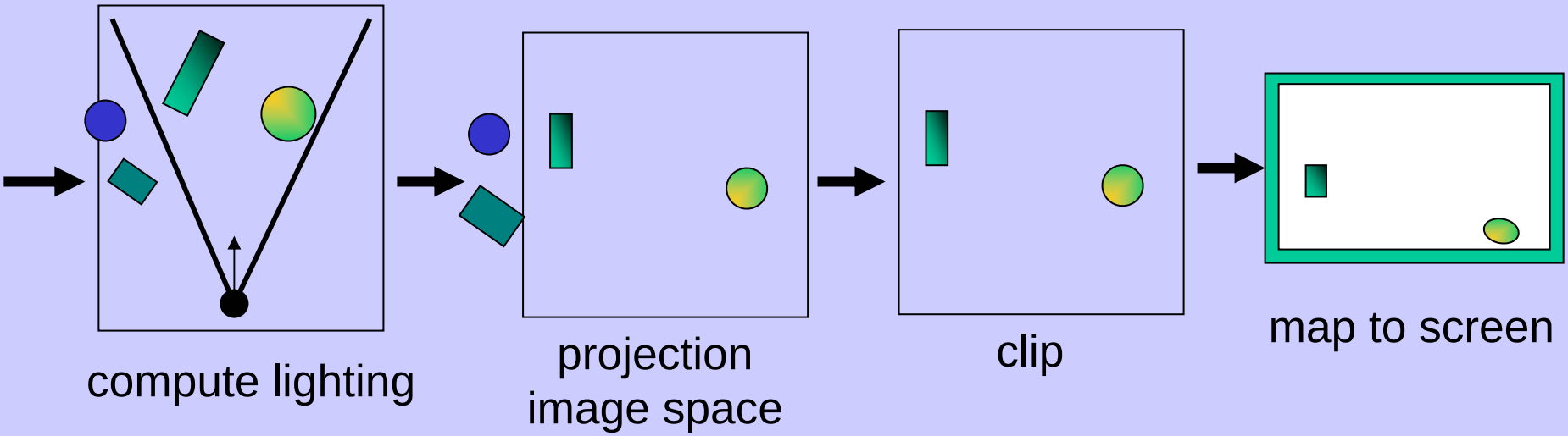
- Square (cube) after projection
- Clip primitives to square

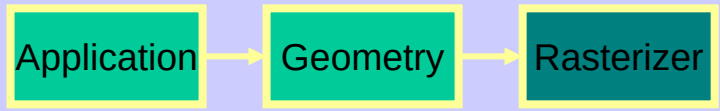


- Screen mapping, scales and translates square so that it ends up in a rendering window
- These ”screen space coordinates” together with Z (depth) are sent to the rasterizer stage

GEOMETRY - Summary





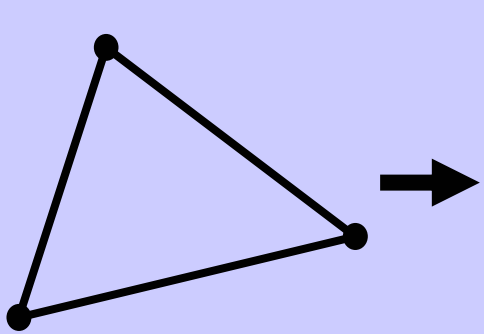


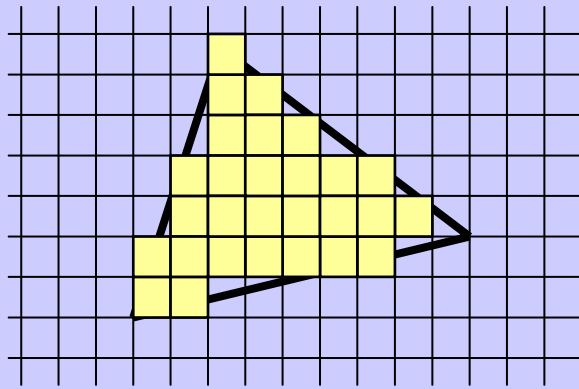
The RASTERIZER in more detail

- 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
- And more…

The RASTERIZER
Scan conversion

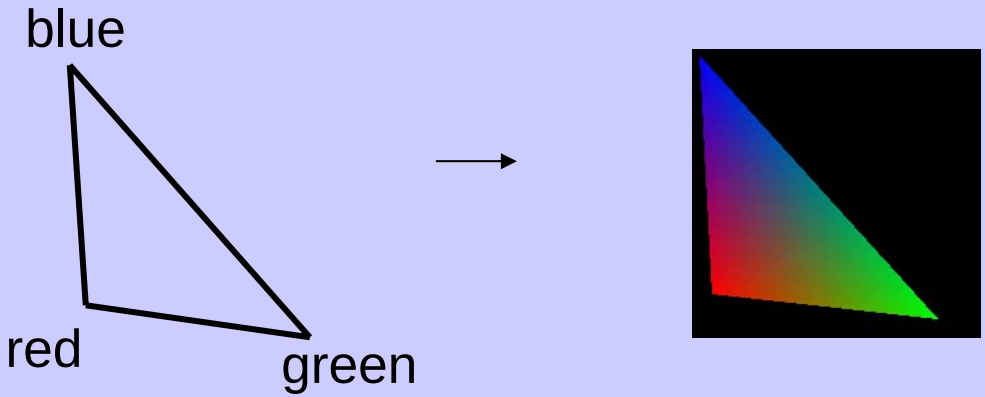
- Triangle vertices from GEOMETRY is input
- Find pixels inside the triangle
  - Or on a line, or on a point
- Do per-pixel operations on these pixels:
  - Interpolation
  - Texturing
  - Z-buffering
  - And more…



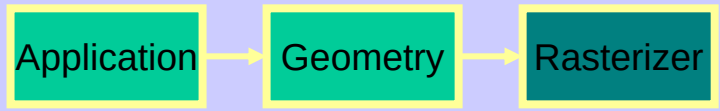


The RASTERIZER Interpolation

- Interpolate colors over the triangle
  - Called Gouraud interpolation

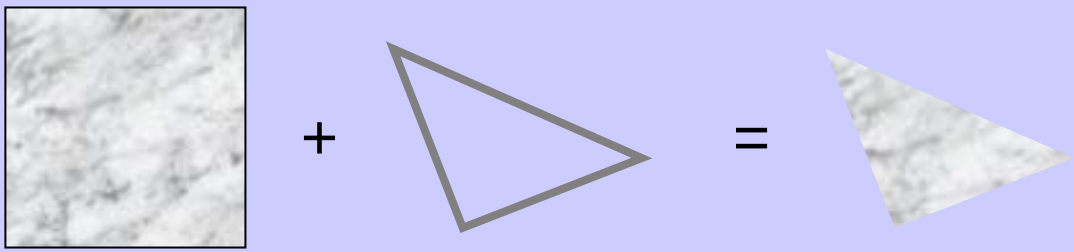


The RASTERIZER



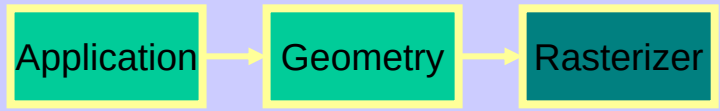
Texturing

- One application of texturing is to ”glue” images onto geometrical object



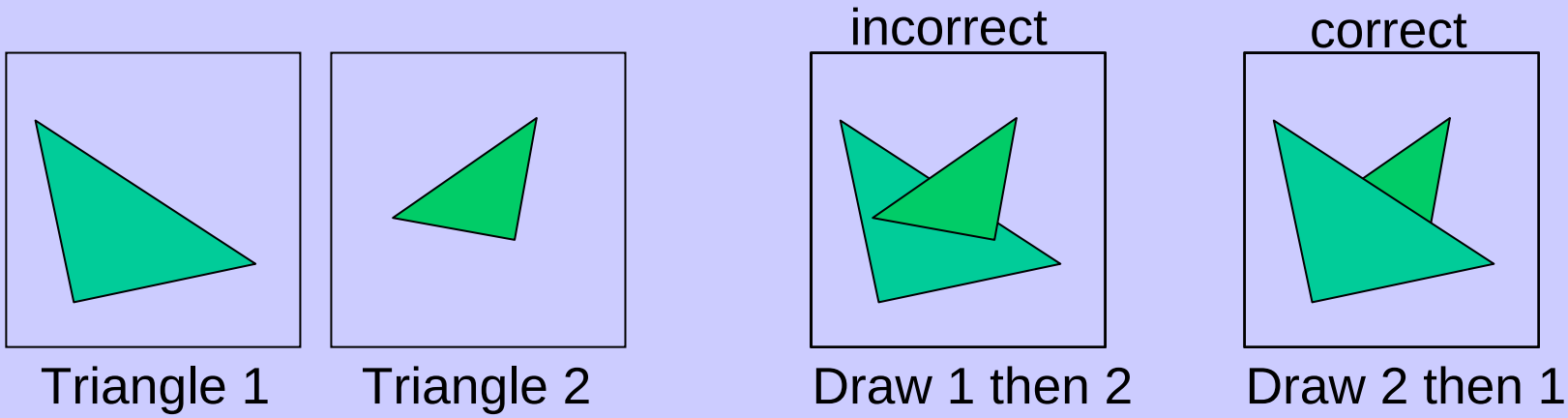
- Uses and other applications
  - More realism
  - Bump mapping
  - Pseudo reflections
  - Store lighting
  - Almost infinitely many uses

The RASTERIZER

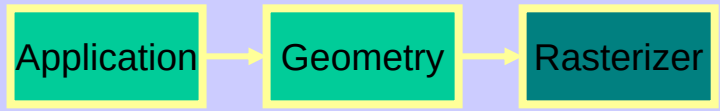


Z-buffering

- The graphics hardware is pretty stupid
  - It ”just” draws triangles
- However, a triangle that is covered by a more closely located triangle should not be visible
- Assume two equally large tris at different depths



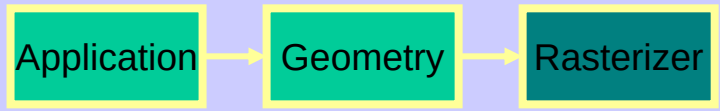
The RASTERIZER



Z-buffering

- Would be nice to avoid sorting…
- The Z-buffer (aka depth buffer) solves this
- Idea:
  - Store z (depth) at each pixel
  - When rasterizing 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

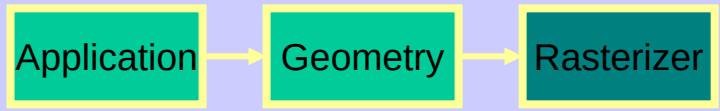
The RASTERIZER double-buffering



- The monitor displays one image at a time
- Top of screen – new image

  Bottom – old image

  No control of split position
- And even worse, we often clear the screen before generating a new image
- A better solution is ”double buffering”
  - (Could instead keep track of rasterpos and vblank).

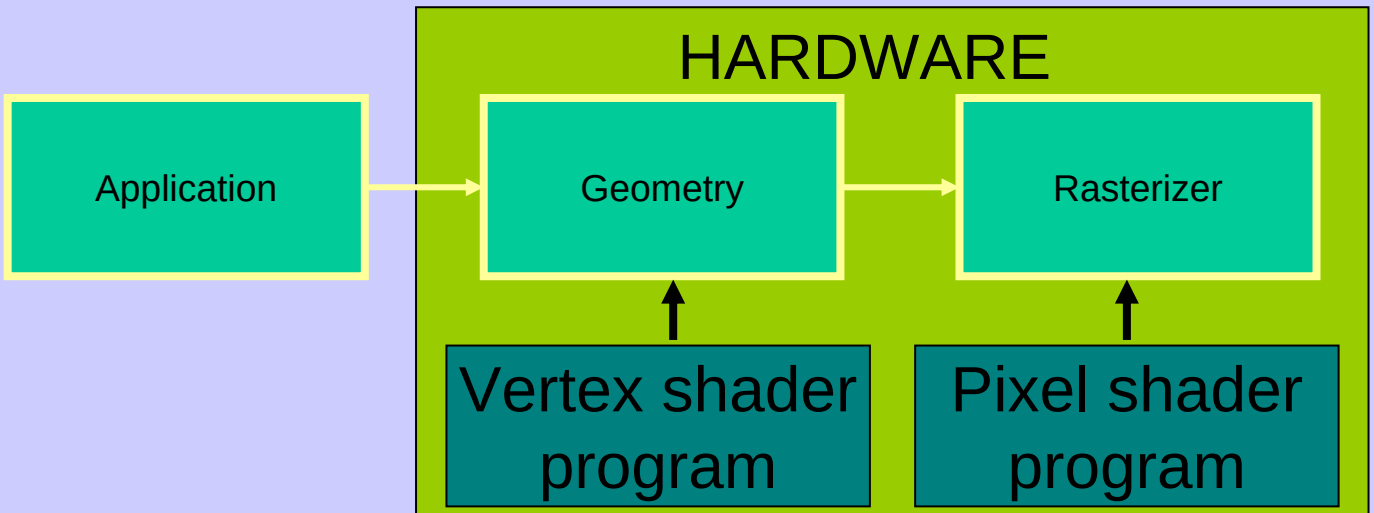


The RASTERIZER
double-buffering

- Use two buffers: one front and one back
- The front buffer is displayed
- The back buffer is rendered to
- When new image has been created in back buffer, swap front and back

And, lately…

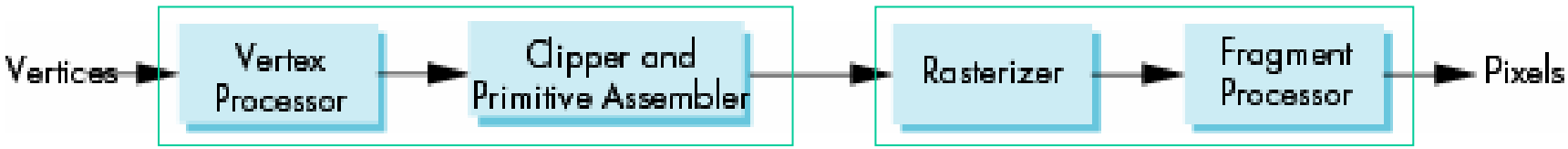
- Programmable shading has become a hot topic
  - Vertex shaders
  - Pixel shaders
  - Adds more control and much more possibilities for the programmer



The Pipeline and OpenGL

Practical Approach

- Process objects one at a time in the order they are generated by the application
  - –Can consider only local lighting
- Pipeline architecture

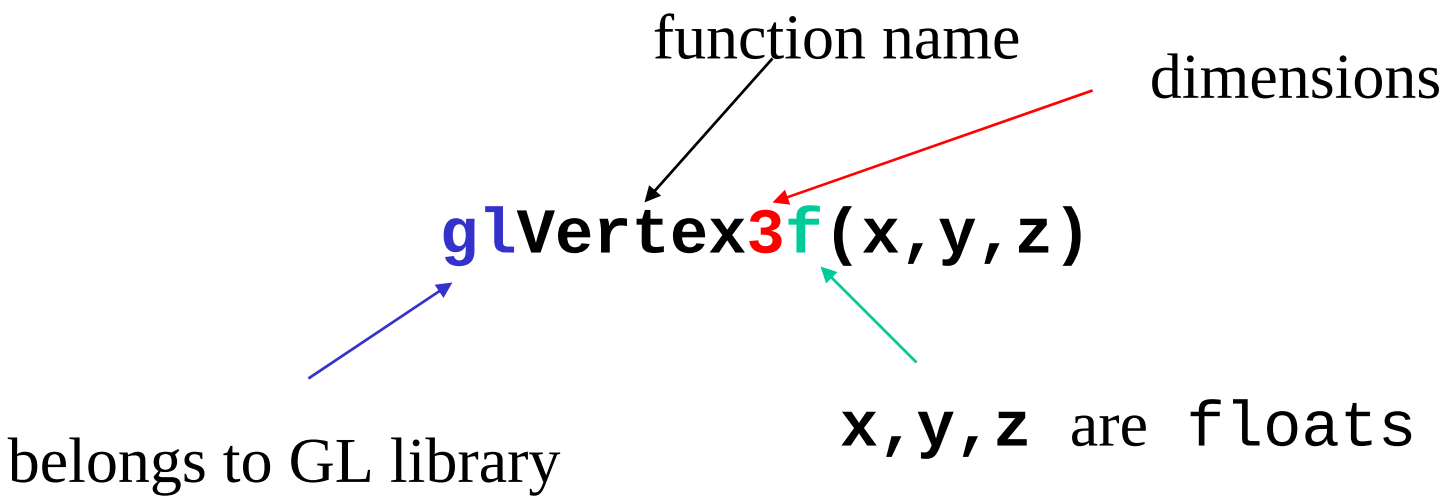


application program

display

- All steps can be implemented in hardware on the graphics card

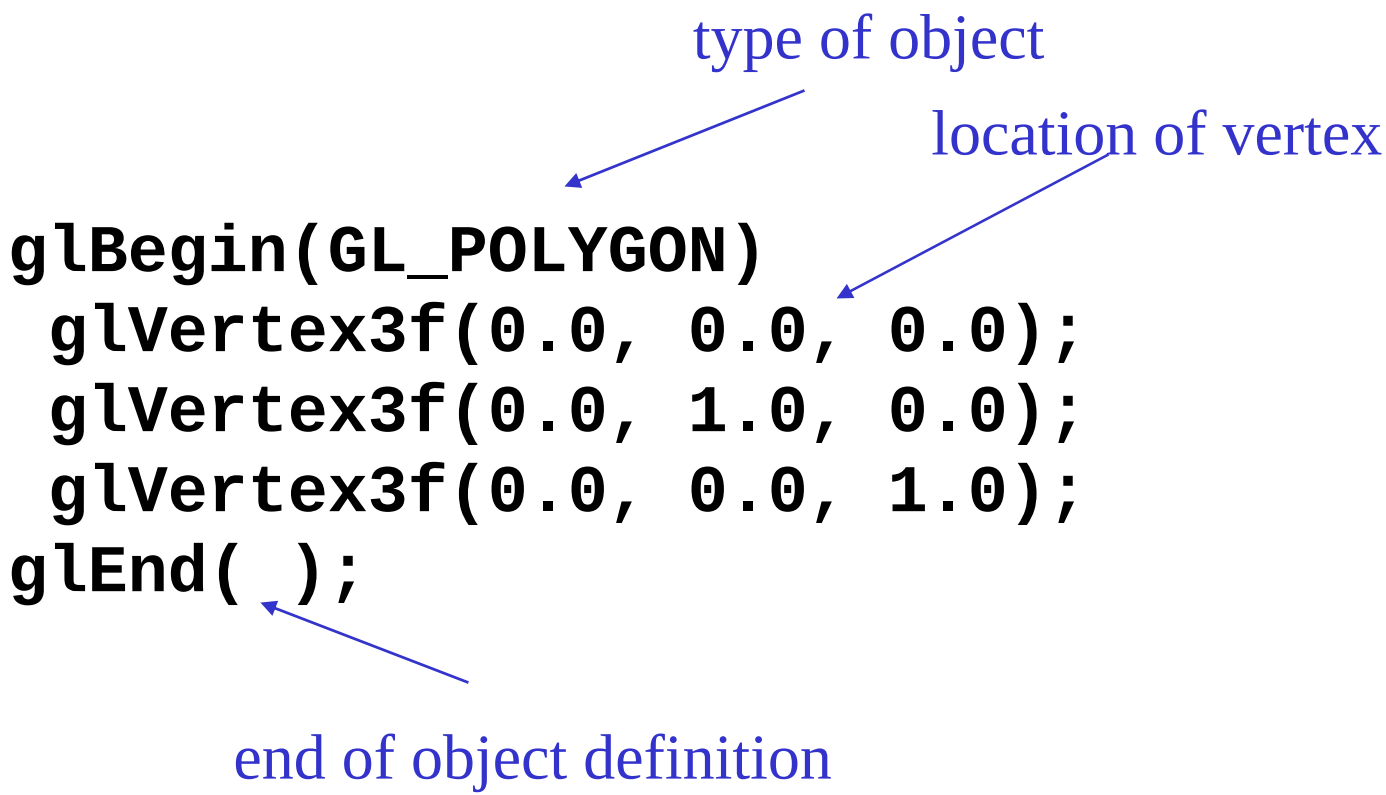
OpenGL function format




```
glVertex3fv(p)
```

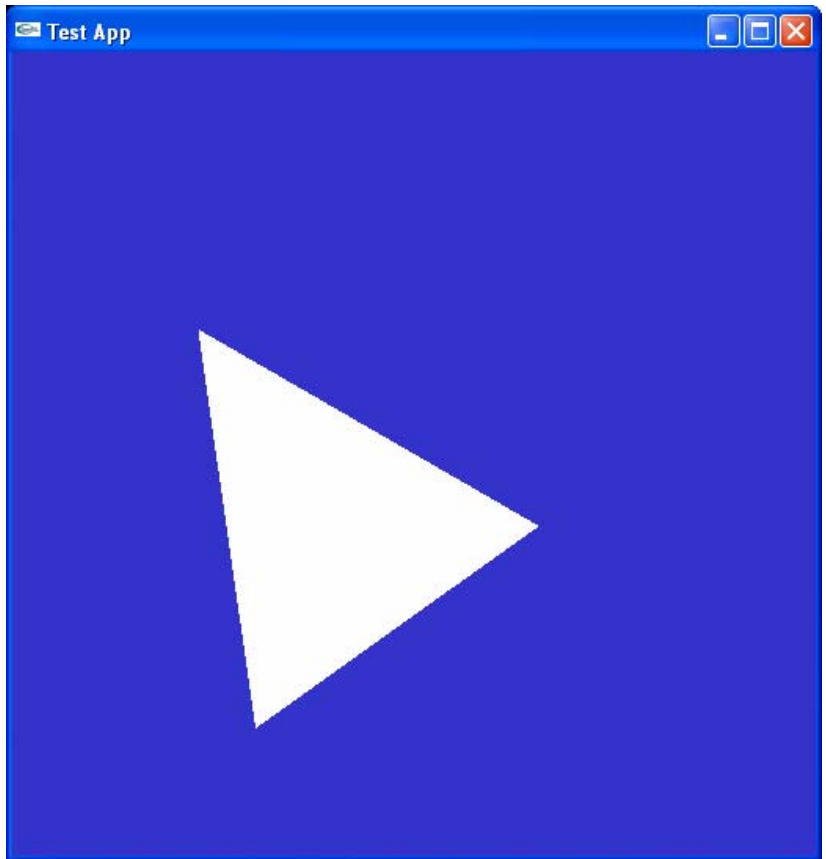
p is a pointer to an array

Example


type of object
location of vertex
glBegin(GL_POLYGON)
glVertex3f(0.0, 0.0, 0.0);
glVertex3f(0.0, 1.0, 0.0);
glVertex3f(0.0, 0.0, 1.0);
glEnd( );
end of object definition

A Simple Program

Generate a triangle on a solid background



Simple Application...

```
#ifdef WIN32
#include <windows.h>
#endif

#include <GL/glut.h>                    // This also includes gl.h

static void drawScene(void)
{
    glColor3f(1,1,1);

    glBegin(GL_POLYGON);
        glVertex3f( 4.0, 0, 4.0);
        glVertex3f( 4.0, 0,-4.0);
        glVertex3f(-4.0, 0,-4.0);
    glEnd();
}
```

Usually this and next 2 slides are put in the same file main.cpp

Simple Application

```
void display(void)
{
    glClearColor(0.2, 0.2, 0.8, 1.0);              // Set clear color
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clears the color buffer
                                                          and the z-buffer
    int w = glutGet((GLenum)GLUT_WINDOW_WIDTH);
    int h = glutGet((GLenum)GLUT_WINDOW_HEIGHT);
    glViewport(0, 0, w, h);                        // Set viewport

    glMatrixMode(GL_PROJECTION);       // Set projection matrix
    glLoadIdentity();
    gluPerspective(45.0,w/h, 0.2, 10000.0);  // FOV, aspect ratio, near, far

    glMatrixMode(GL_MODELVIEW);        // Set modelview matrix
    glLoadIdentity();

    gluLookAt(10, 10, 10,                          // look from
              0, 0, 0,                             // look at
              0, 0, 1);                            // up vector

    drawScene();
    glutSwapBuffers();  // swap front and back buffer. This frame will now been displayed.
}
```

Simple Application...

```
int main(int argc, char *argv[])
{
    glutInit(&argc, argv);

    /* open window of size 512x512 with double buffering, RGB colors, and Z-
    buffering */
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize(512,512);
    glutCreateWindow("Test App");

    /* the display function is called once when the gluMainLoop is called,
     * but also each time the window has to be redrawn due to window
     * changes (overlap, resize, etc).  */
    glutDisplayFunc(display);          // Set the main redraw function

    glutMainLoop();  /* start the program main loop */
    return 0;
}
```

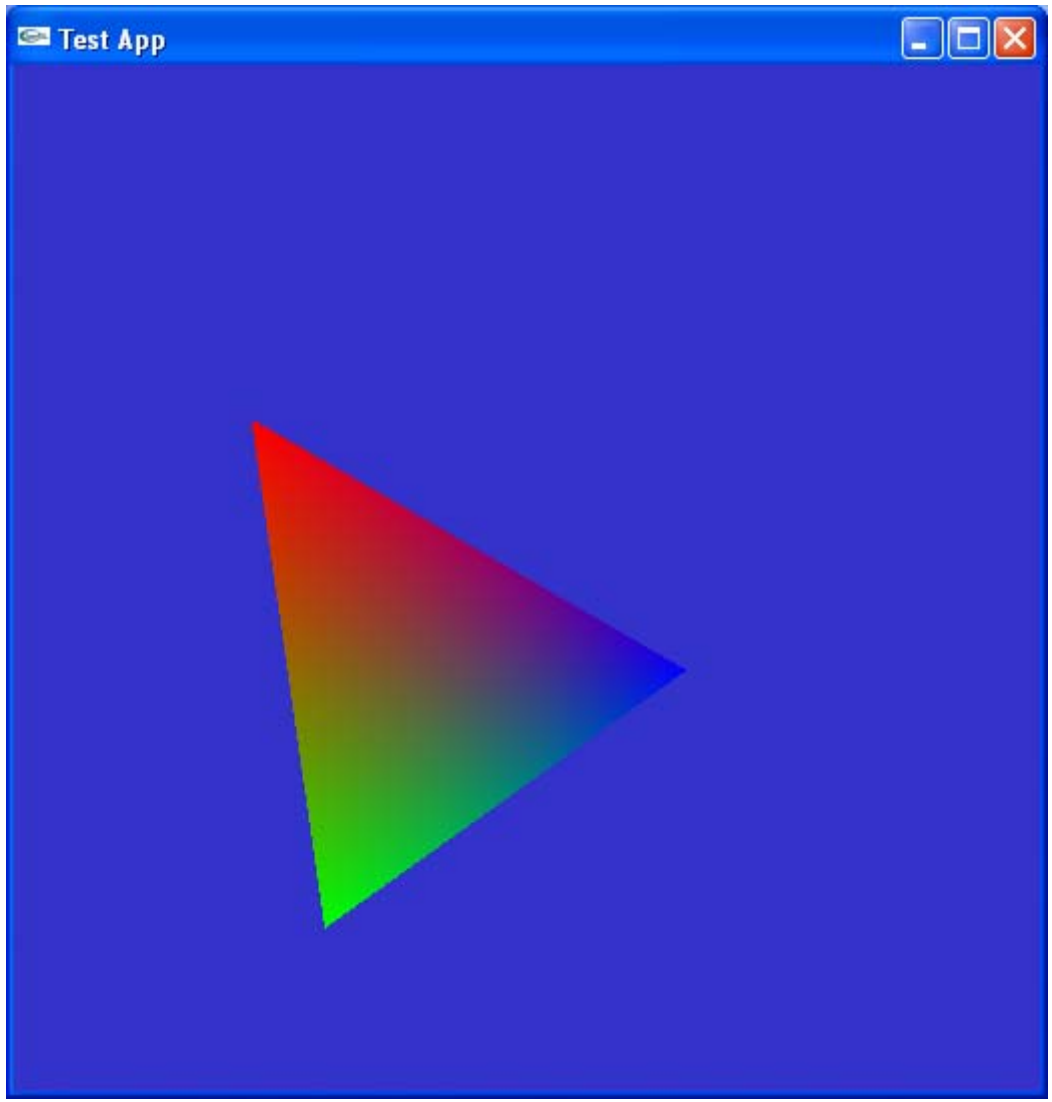
Changing Color per Vertex

```
static void drawScene(void)
{
    // glColor3f(1,1,1);
    glBegin(GL_POLYGON);
        glColor3f(1,0,0);
        glVertex3f( 4.0, 0, 4.0);

        glColor3f(0,1,0);
        glVertex3f( 4.0, 0,-4.0);

        glColor3f(0,0,1);
        glVertex3f(-4.0, 0,-4.0);
    glEnd();
}
```

Cooler application



Questions

- If we have time
  - Hardware history
  - OpenGL history
- Else See you Tuesday 8-10
  - The maths (transformations)

Hårdvaruhistoria

Datorgrafik - utveckling

Återgivning

- Streckfigurer (tråddiagram)
- Fyllda ytor (dolda ytproblemet)
- Texturerade ytor
- Fotorealism (ljus, skugga, reflektioner)
- Animering
- Ljud
- Multimedia
- Nätbaserat
- I mobiltelefon el dyl, stereo, VR

Interaktion
Alltid varit centralt. Men utrustningen blivit allt mer sofistikerad.
Haptik (återkoppling - feedback - i form av virtuell känsla).
http://haptic.mech.northwestern.edu/intro/gallery/

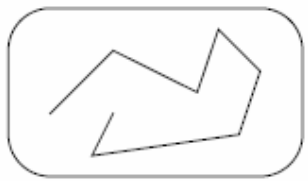
Hårdvara genom tiderna

Presentationsutrustning

Analog teknik

Skrivare: Pennplottrar (kurvritare)

Skärm: Oscilloskopteknik, t ex Tektronix. Bilden (bara den) ritas oupphörligt.

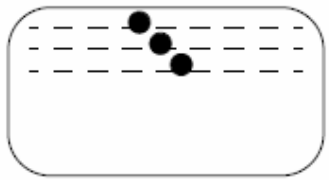


1980

Digital teknik

Skrivare: Laserskrivare (1986)
Bläckstråleskrivare

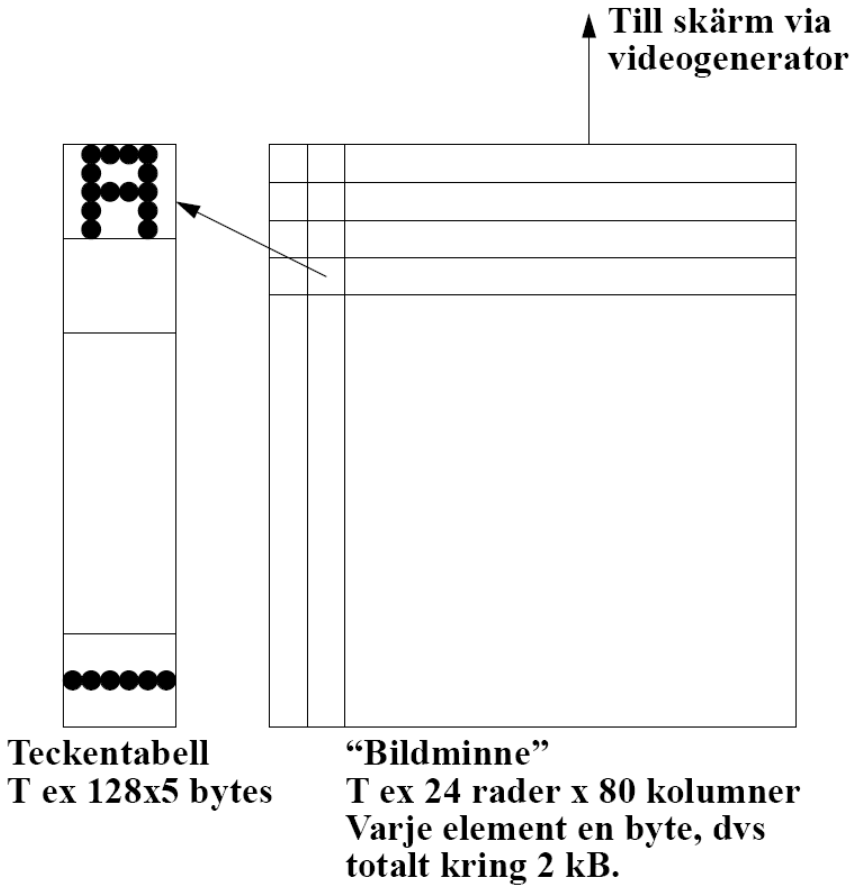
Skärm: TV-teknik (videosignal från bildminne)



Tid

Jfr Håkan Lans patentstrider (patent inlämnat 1979). Men det var mycket sådant i luften då.

Datorgrafik - textterminaler kring 1980



- S k semigrafik (motsv Text-TV) möjlig.
- C64 fungerade på detta sätt (1982-91).
- Numera är alla datorer "grafiska".

Direct View Storage Tube

- Created by Tektronix
  - Did not require constant refresh
  - Standard interface to computers
    - Allowed for standard software
    - Plot3D in Fortran
  - Relatively inexpensive
    - Opened door to use of computer graphics for CAD community

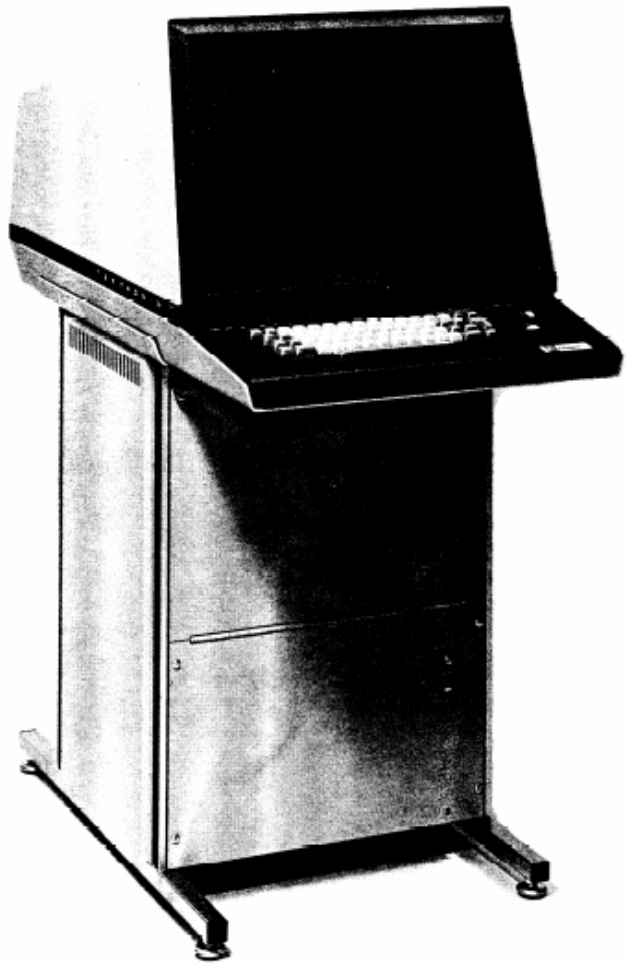
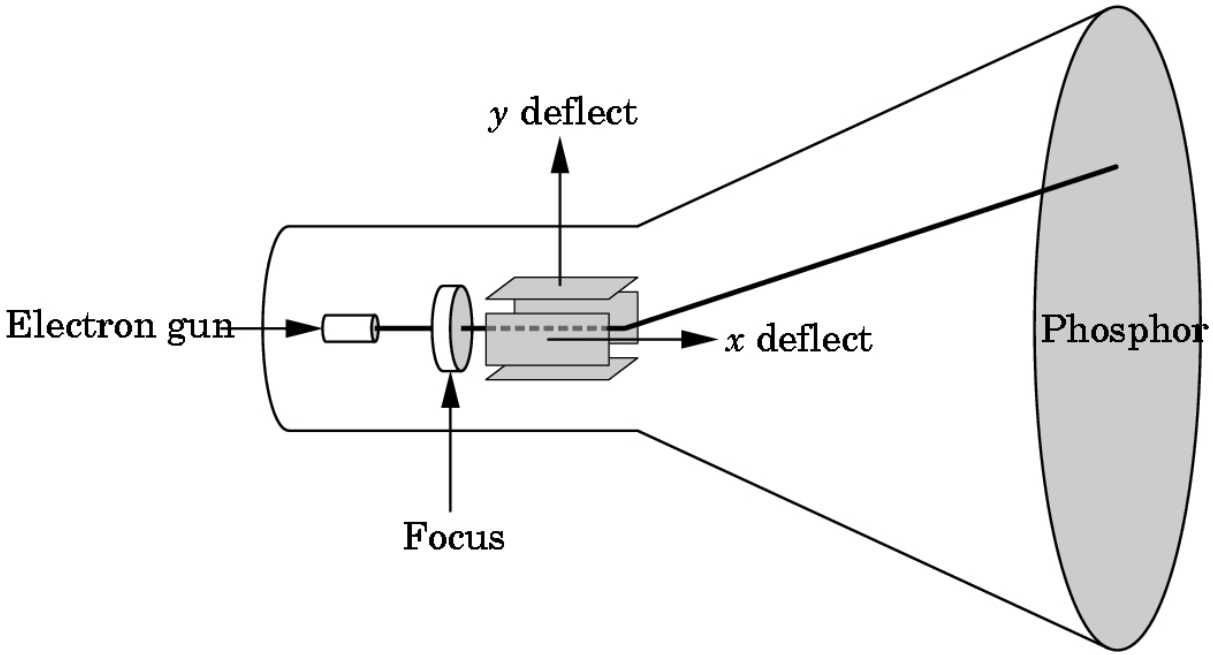


Fig. 1-1. 4014 Computer Display Terminal.

CRT

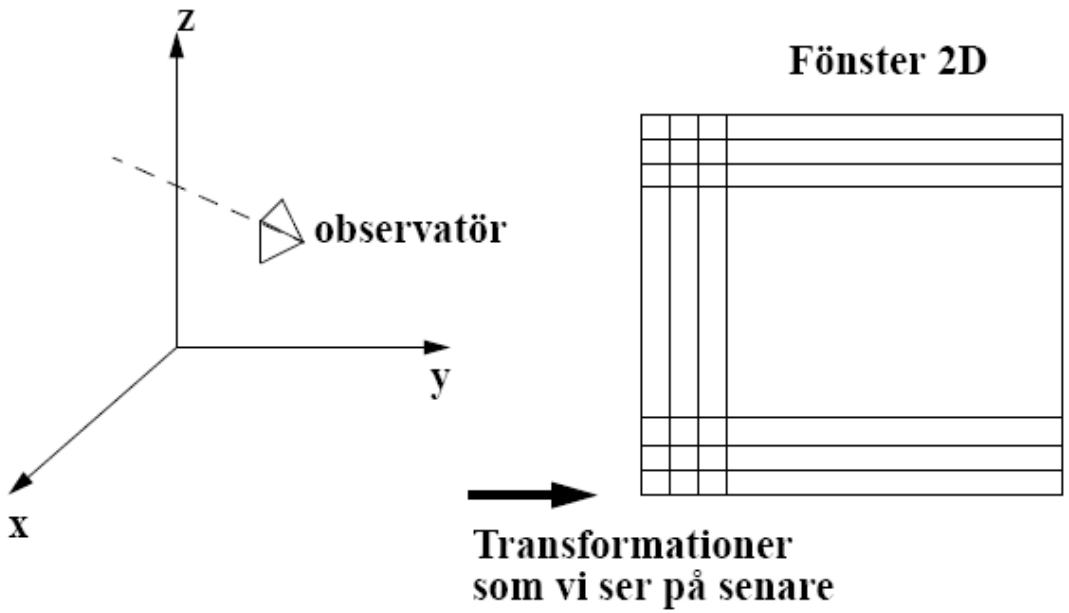


Can be used either as a line-drawing device (calligraphic) or to display contents of frame buffer (raster mode)

Graphics Hardware History

- 80's:
  - linear interpolation of color over a scanline
  - Vector graphics
- 91' Super Nintendo, Neo Geo,
  - Rasterization of 1 single 3D rectangle per frame (FZero)
- 95-96': Playstation 1, 3dfx Voodoo 1
  - Rasterization of whole triangles
- 99' Geforce (256)
  - Transforms and Lighting (geometry stage)
- 02' 3DLabs WildCat Viper, P10
  - Pixel shaders, integers,
- 02' ATI Radion 9700, GeforceFX
  - Vertex shaders and **Pixel shaders** with floats

3D-Grafik

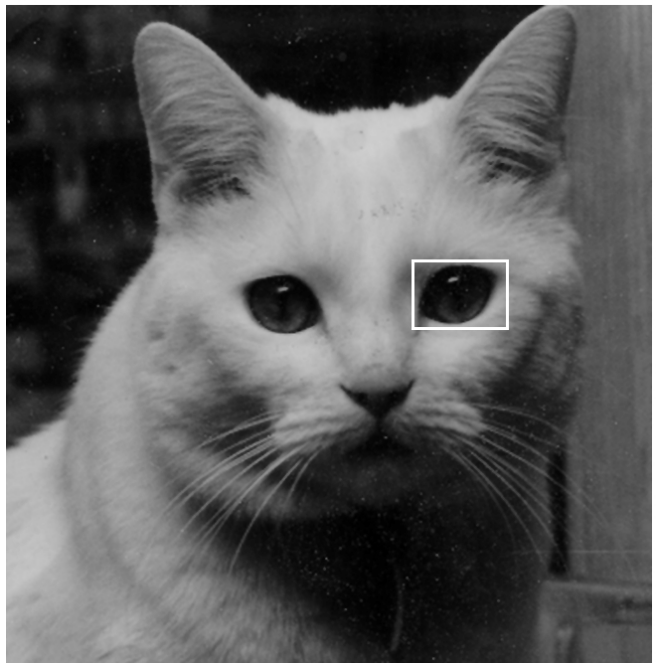


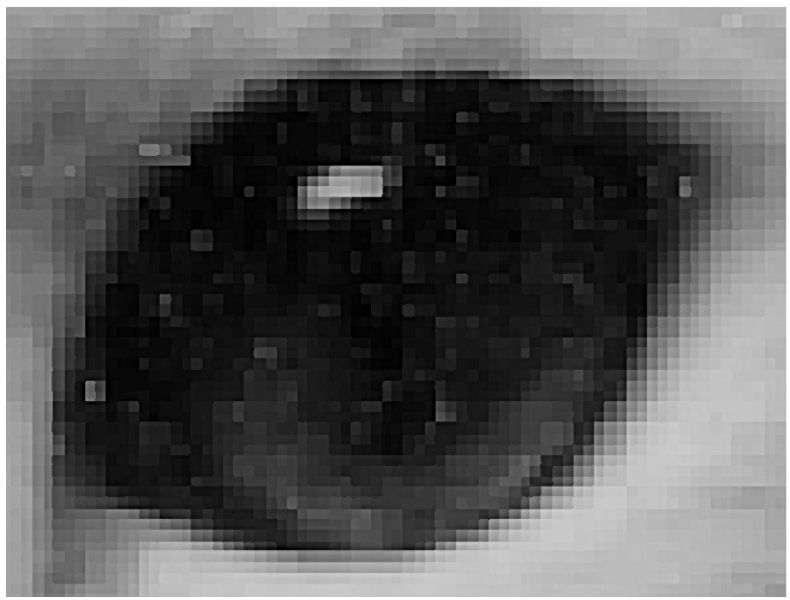
Värld med
- föremål
- observatör (kamera)

Vad behövs?
- observatörens position
- Tittriktning
- Uppåtriktning
- Synfält (vinkel o dyl)

Raster Graphics

- Image produced as an array (the *raster*) of picture elements (*pixels*) in the *frame buffer*





Datorgrafik - hårdvara

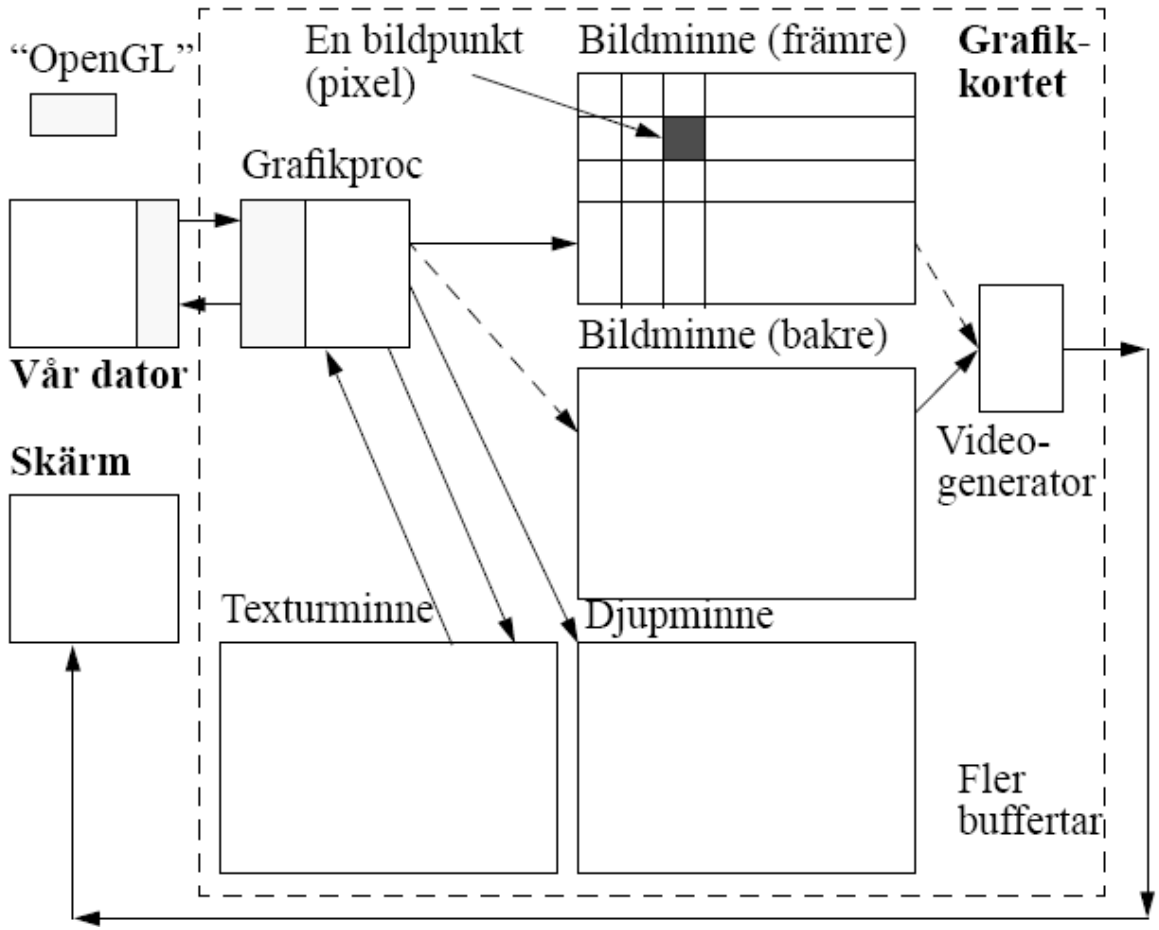
Så här fungerar det i stort i ett system med skärm som utenhet:

• Typiskt skickar vår dator punkter, linjeändpunkter eller triangelhörn till GPU'n som rastrerar och fyller med färg.

• Uppdatering till skärm sker från bildminnet gärna minst 60 ggr/sekund.

• En modern GPU klarar (i princip) 0.1 - 1 miljarder trianglar per sekund. I praktiken några miljoner trianglar dynamiskt i realtid. Kräver även snabb CPU. Och ofta plattformsberoende kodning!

• Bildminnet kräver storleksordningen 1000x1000x4 bytes = 4 MB. Ingår ofta i texturminnet på 64-512 MB.

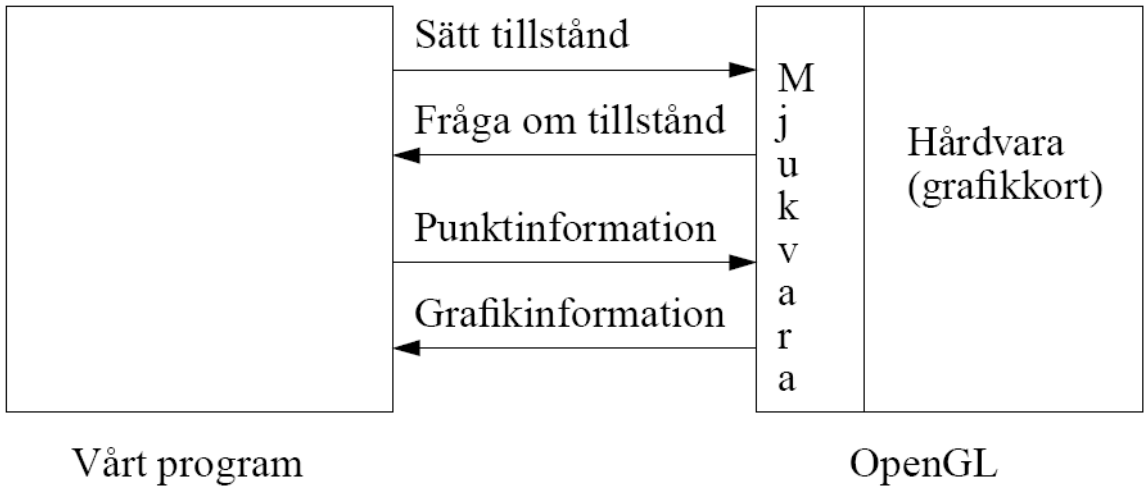


OpenGL i korthet

Referens: Häftet “Introduktion till OpenGL”

OpenGL är

- ett bibliotek för 3D-grafik
- en tillståndsmaskin
- en pipelinemaskin, som i mer eller mindre hög grad kan implementeras i hårdvara
- plattformsoberoende (men språk som använder OpenGL är det inte)



Konkurrent: Microsofts multimediasystem DirectX (inkluderar Direct 3D), nu i version 9. Konkurrensen välgörande. DirectX bara för Windows, medan OpenGL för SGI, Sun Solaris, Linux, Windows, Mac, ...

HISTORY OF OPENGL

SGI and GL

- Silicon Graphics (SGI) revolutionized the graphics workstation by implementing the pipeline in hardware (1982)
- To access the system, application programmers used a library called GL
- With GL, it was relatively simple to program three dimensional interactive applications

OpenGL

The success of GL lead to OpenGL (1992), a platform-independent API that was

- Easy to use
- Close enough to the hardware to get excellent performance
- Focus on rendering
- Omitted windowing and input to avoid window system dependencies

OpenGL Evolution

- Controlled by an Architectural Review Board (ARB)
  - –Members include SGI, Microsoft, Nvidia, HP, 3DLabs, IBM,…….
  - –Relatively stable (present version 2.0)
    - Evolution reflects new hardware capabilities
      - **3D texture mapping and texture objects**
      - **Vertex programs**
  - –Allows for platform specific features through extensions

OpenGL Libraries

- OpenGL core library
  - –OpenGL32 on Windows
  - –GL on most unix/linux systems (libGL.a)
- OpenGL Utility Library (GLU)
  - –Provides functionality in OpenGL core but avoids having to rewrite code
- Links with window system
  - –GLX for X window systems
  - –WGL for Windows
  - –AGL for Macintosh

GLUT

- OpenGL Utility Toolkit (GLUT)
  - Provides functionality common to all window systems
    - Open a window
    - Get input from mouse and keyboard
    - Menus
    - Event-driven
  - Code is portable but GLUT lacks the functionality of a good toolkit for a specific platform
    - No slide bars

Software Organization