TDA361 – Computer Graphics

Teacher: **Ulf Assarsson**

Chalmers University of Technology
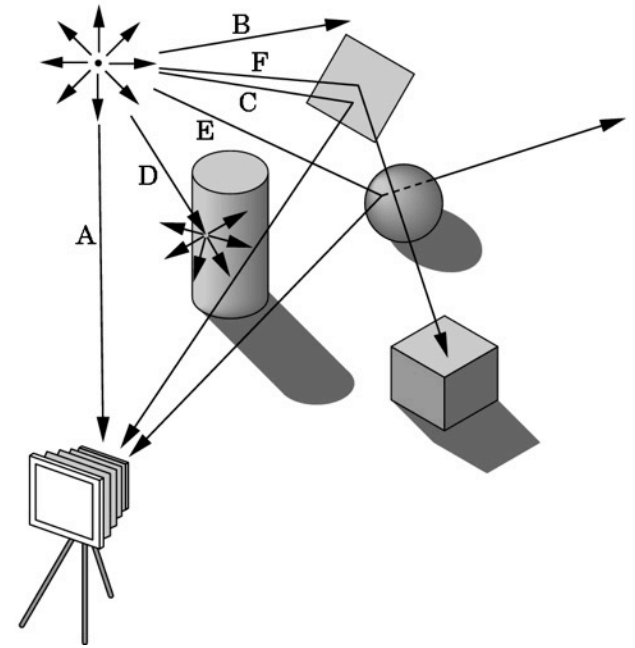
This Course

- Algorithms!

Real-time Rendering

Understanding of Ray Tracing

Tracing Photons

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.
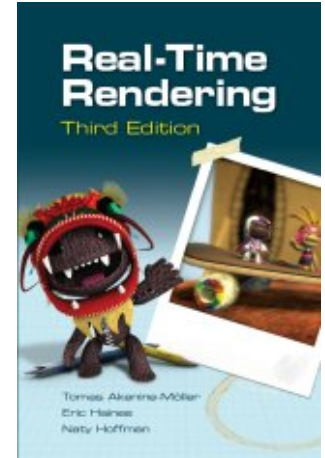
Other Physical Approaches

- **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
  - Faster but still slow
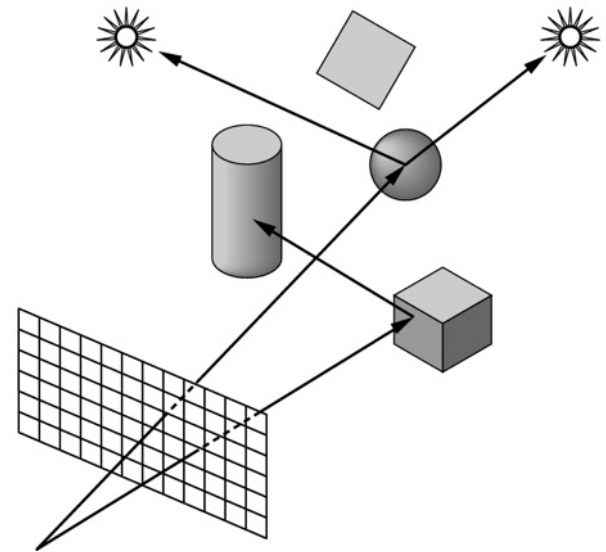
Course Info

- Study Period 2 (lp2)
- Real Time Rendering, 3rd edition
  - Available on Cremona
- Homepage:
 - Google "TDA361" or
  - "Computer Graphics Chalmers"
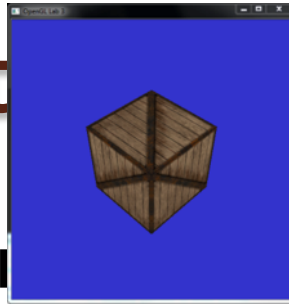
Tutorials

- All laborations are in C++ and OpenGL
  - Industry standard
  - No previous (C++) knowledge required
- Six shorter tutorials that go through basic concepts
  - Basics, Textures, Camera&Animation, Shading, Render-to-texture, Shadow Mapping
- One slightly longer lab where you put everything together
  - Render engine (e.g for a game)
  - or
  - Path tracer
  - (Or own project)
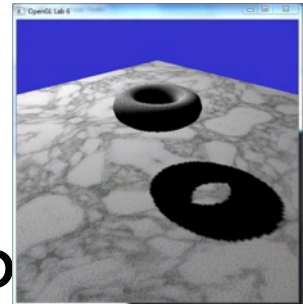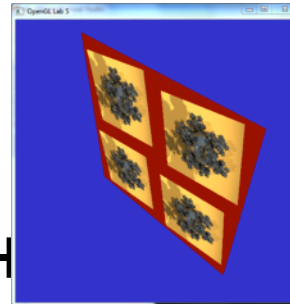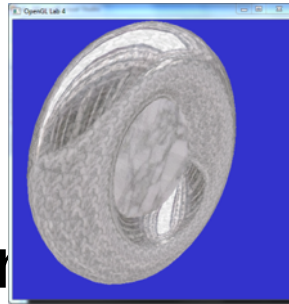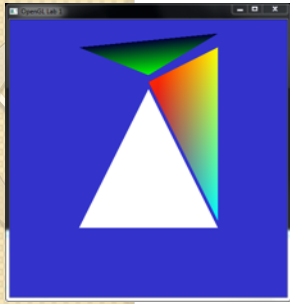
Tutorials

- Rooms 4225
  - Or your favorite place/home
- 4th floor EDIT-building
- EntranceCards (inpasseringskort)
  - Automatically activated for all of you that are course registered and have a CTH/GU-entrance card (inpasseringskort)
- Recommended to do the tutorials in groups (Labgrupper) of 2, or individually if you prefer.

Lab Partner List

- Do the tutorials individually or in pairs.

- If you want a lab partner
  - Write your name + email on list at desk in the break.

LAB PARTNER LIST

Sign up at a free slot, if you do not already have a lab partner. You can also do the tutorials by yourself – in which case you do not need to sign up below.

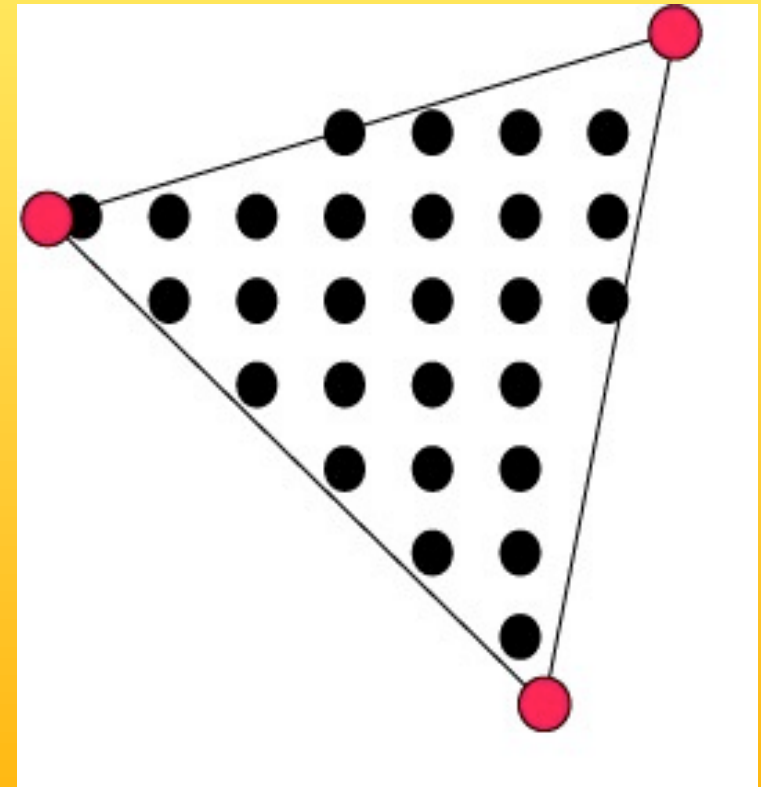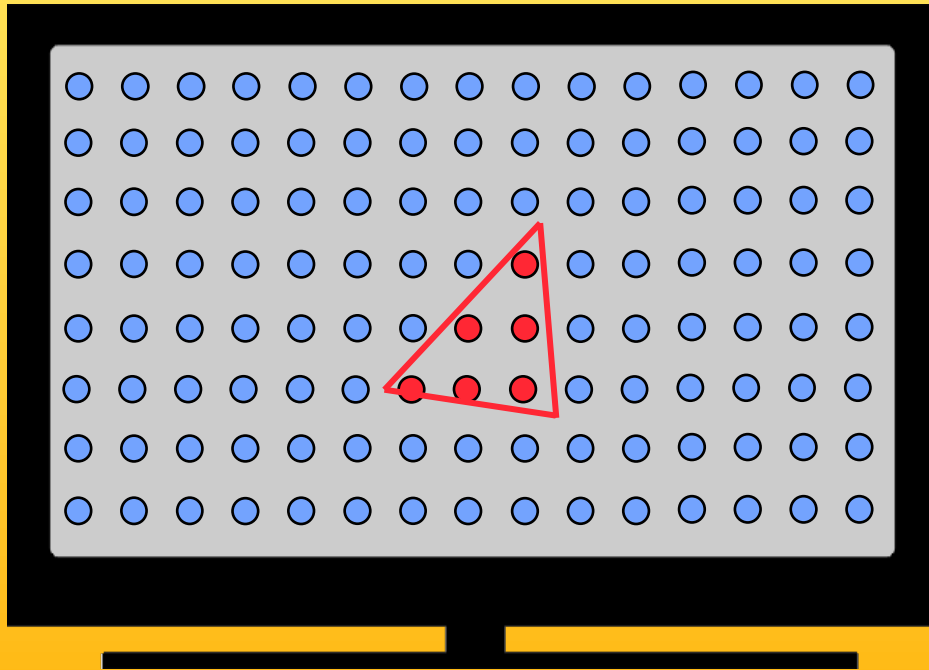# Overview of the Graphics Rendering Pipeline and OpenGL

CHALMERS

3D Graphics

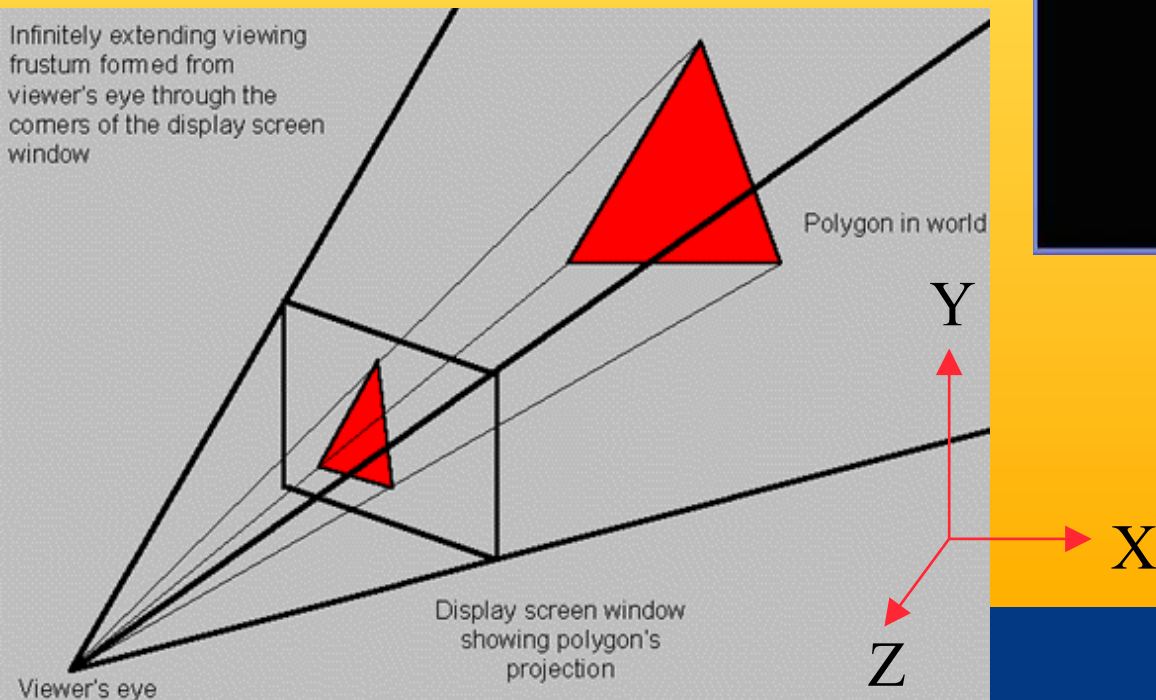Ulf Assarsson

The screen consists of pixels

3D-Rendering

- Objects are often made of triangles

- x,y,z- coordinate for each vertex

Infinitely extending viewing frustum formed from viewer's eye through the corners of the display screen window

Polygon in world

Display screen window showing polygon's projection

Viewer's eye

Y

X

Z

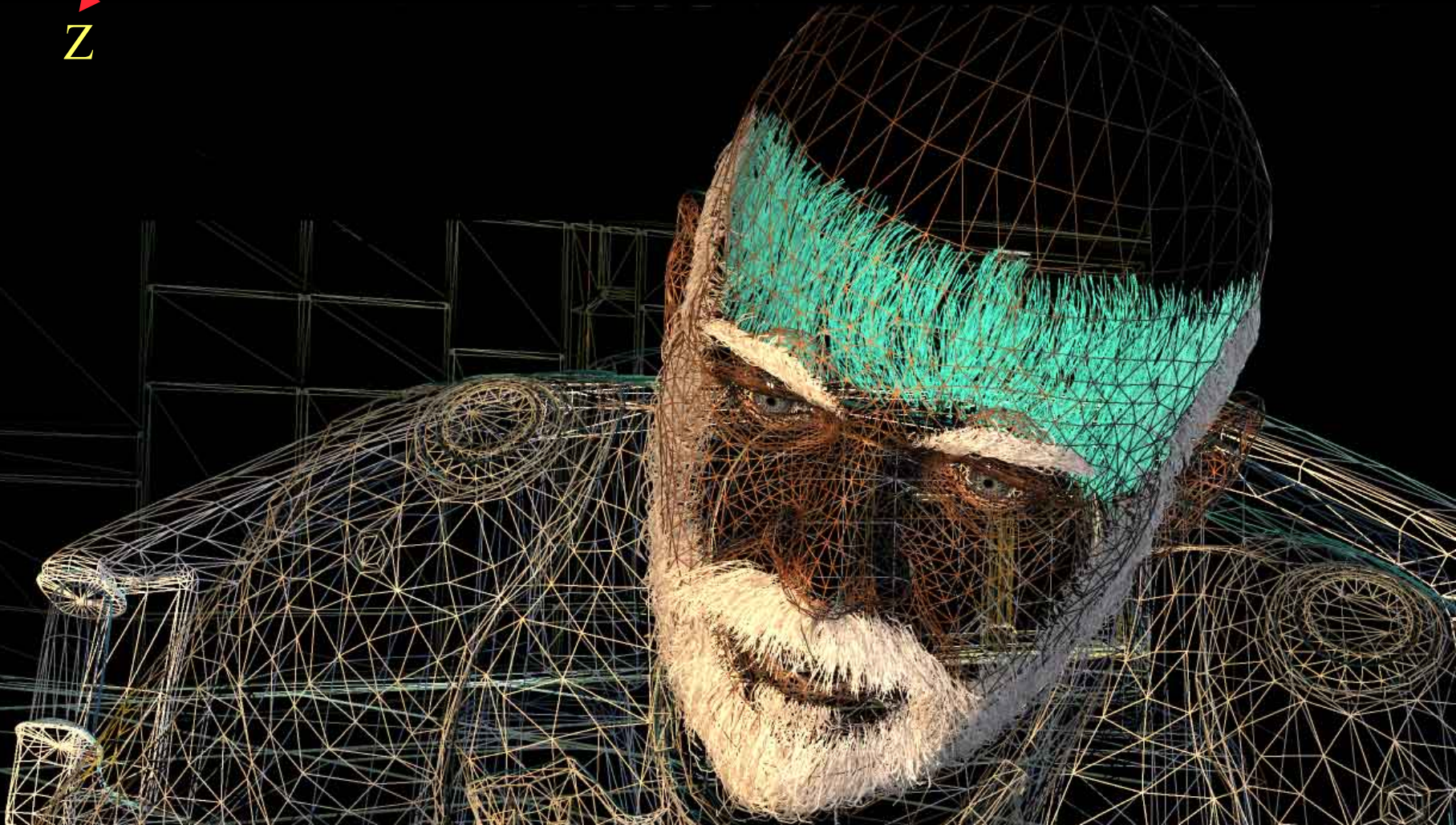(C) 1998 Evans & Sutherland Glaze v3.1
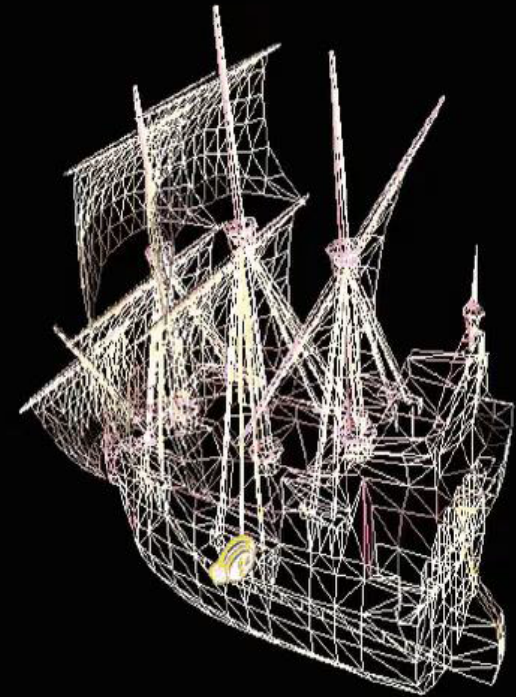
Grafikkort
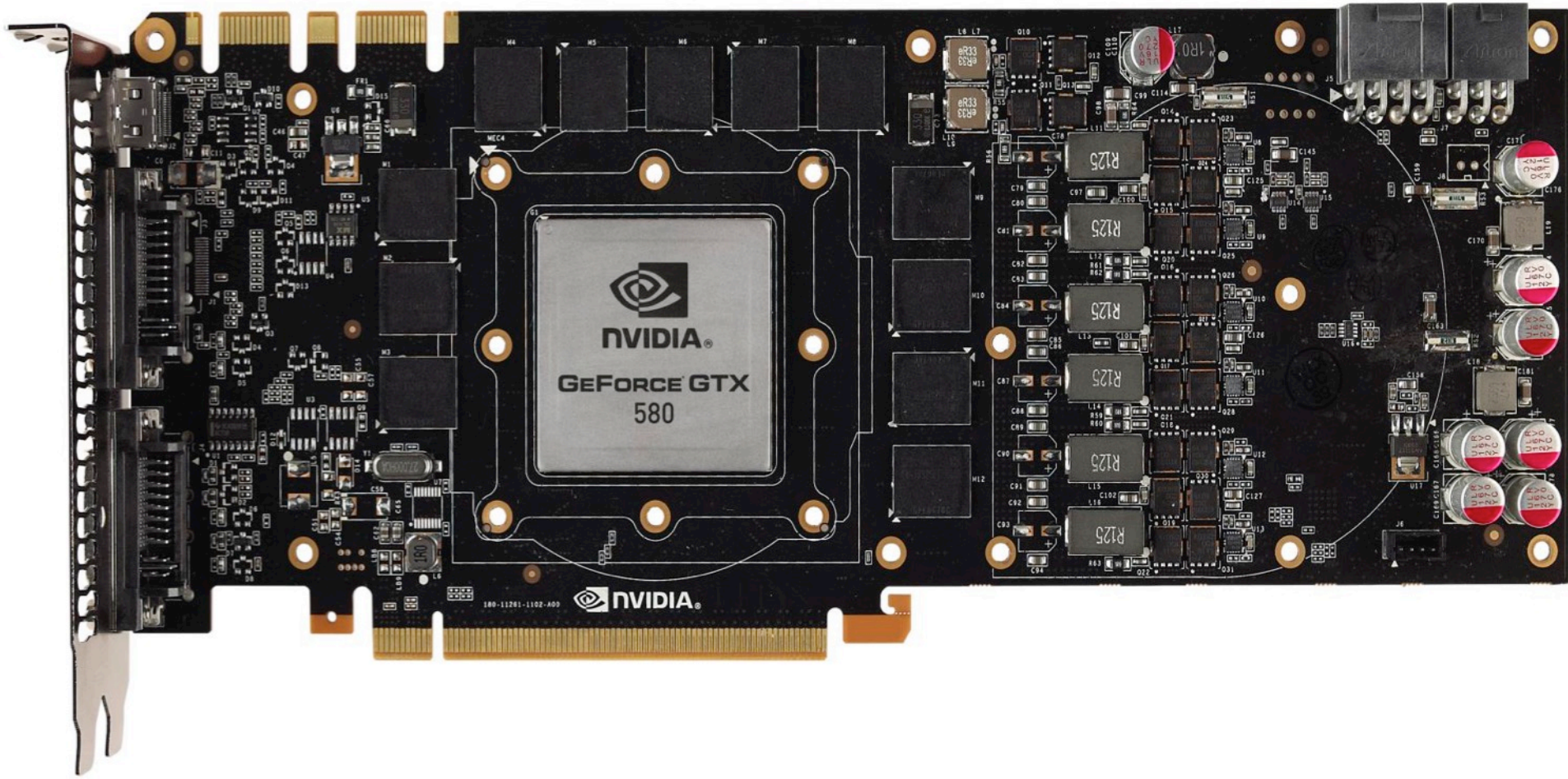
4D Matrix Multiplication

$$\begin{bmatrix} s_x & \bullet & \bullet & t_x \\ \bullet & s_y & \bullet & t_y \\ \bullet & \bullet & s_z & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

Real-Time Rendering
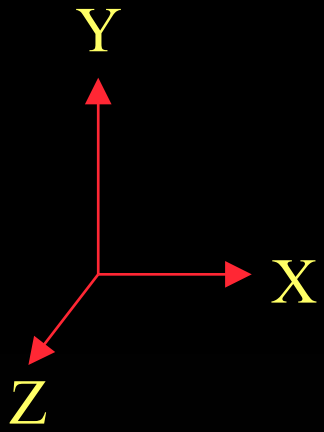
Textures

- 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
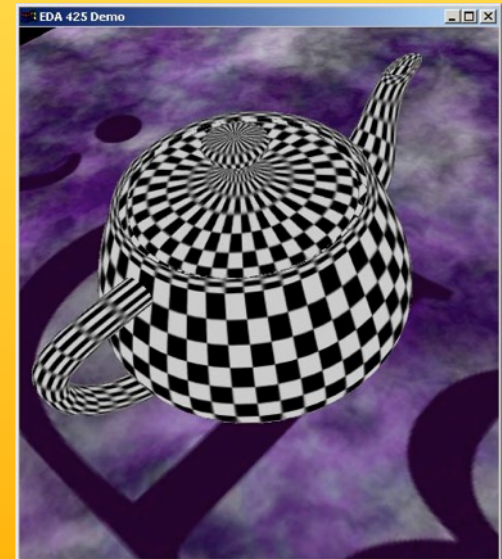
Lighting computation per triangle vertex
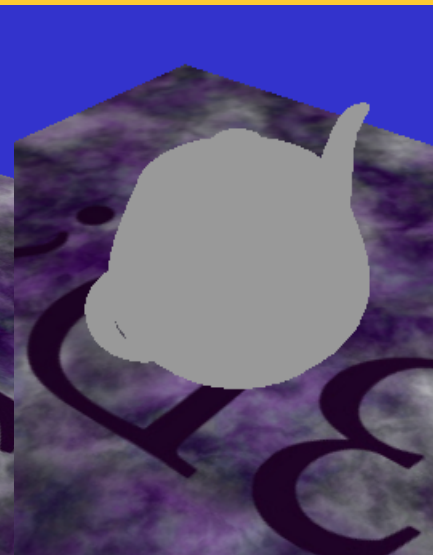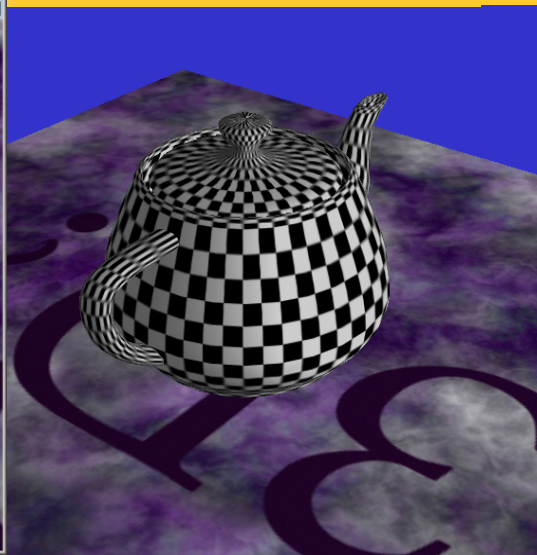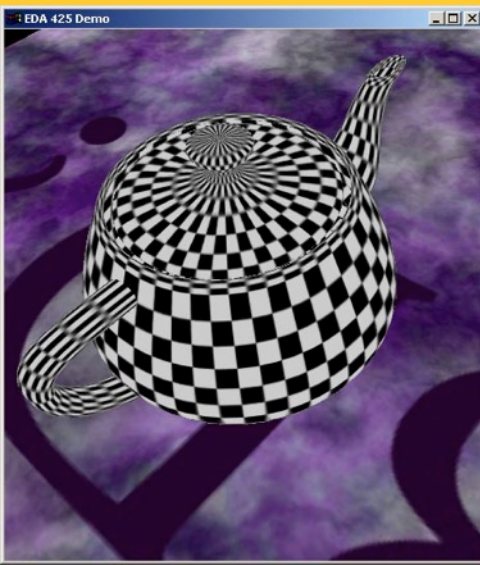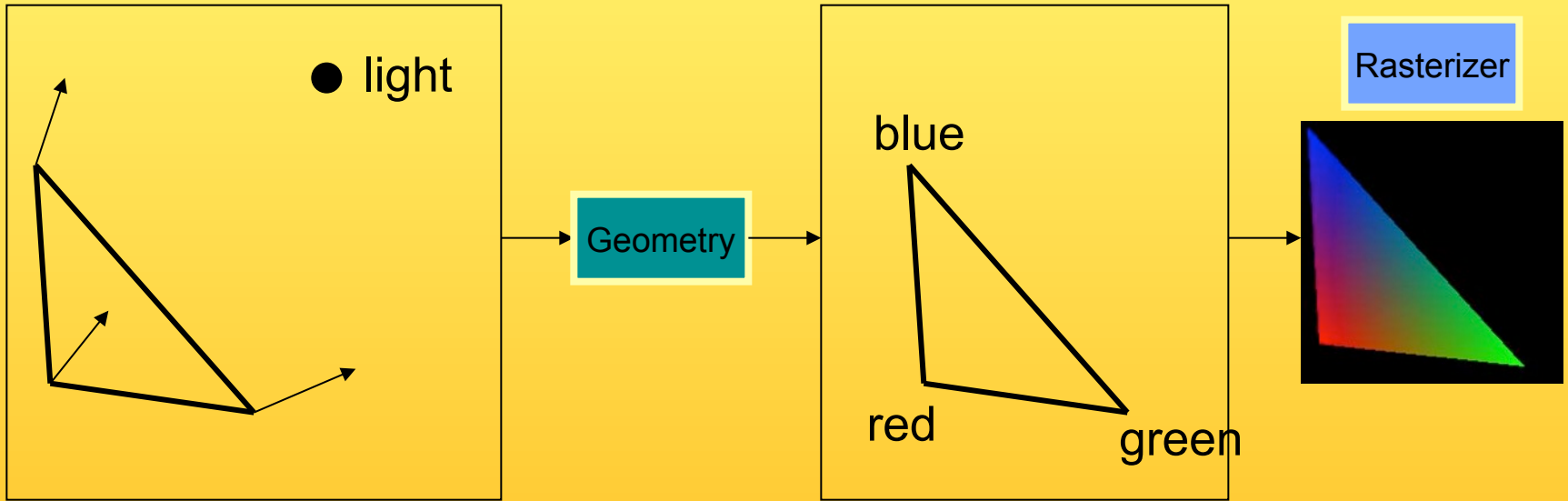
The Graphics Rendering Pipeline

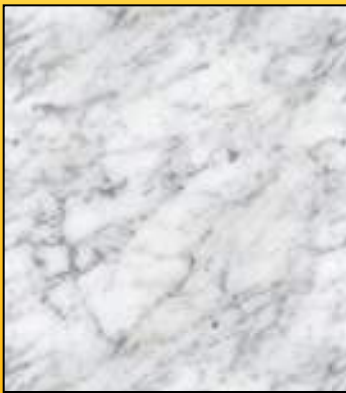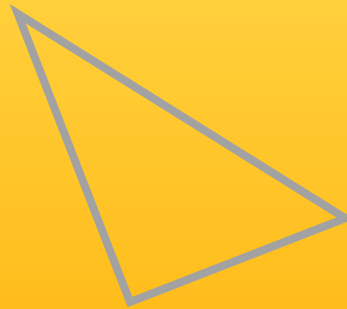# You say that you render a *"3D scene"*, but what is it?

- 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
    - Colors, shader code ,
    - Textures (images to glue onto the geometry)
- A triangle consists of 3 vertices
  - A vertex is 3D position, and may include normals.
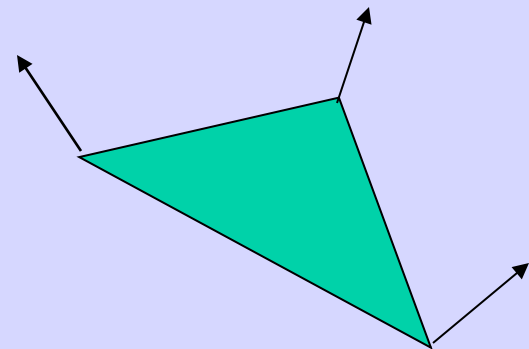
Lecture 1: Real-time Rendering
The Graphics Rendering Pipeline

- The pipeline is the "engine" that creates images from 3D scenes

- Three conceptual stages of the pipeline:
  - Application (executed on the CPU)
  - Geometry
  - Rasterizer

Application → Geometry → Rasterizer → Image

input 3D scene

output
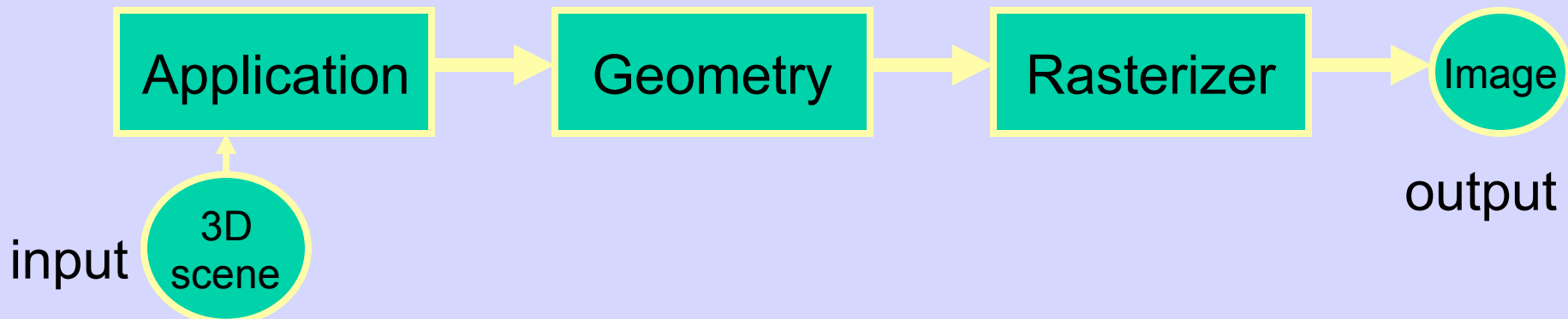
The APPLICATION stage

- Executed on the CPU
  - Means that the programmer decides what happens here
- Examples:
  - Collision detection
  - Speed-up techniques
  - Animation
- Most important task: feed geometry stage with the primitives (e.g. triangles) to render

The GEOMETRY stage

Infinitely extending viewing frustum formed from viewer's eye through the corners of the display screen window

Polygon in world

Display screen window showing polygon's projection

Viewer's eye

- Task: "geometrical" operations on the input data (e.g. triangles)

- Allows:
  - Move objects (matrix multiplication)
  - Move the camera (matrix multiplication)
  - Lighting computations per triangle vertex
  - Project onto screen (3D to 2D)
  - Clipping (avoid triangles outside screen)
  - Map to window
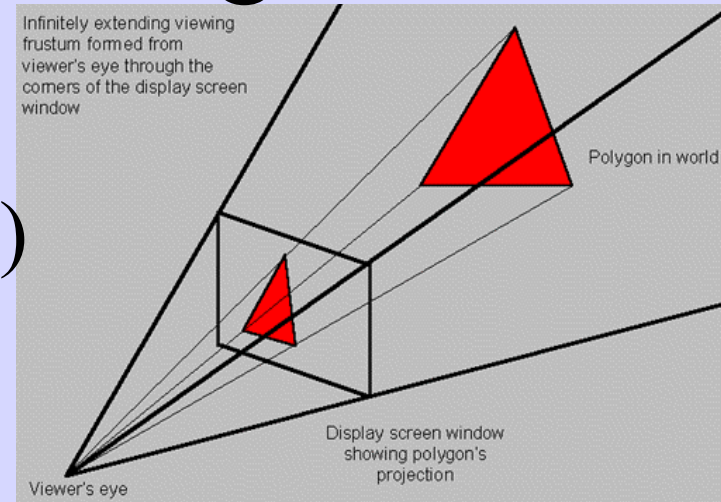
The GEOMETRY stage

| Model & View Transform | → | Vertex Shading | → | Projection | → | Clipping | → | Screen Mapping |

- (Instances)
- Vertex Shader
  - A program executed per vertex
    - Transformations
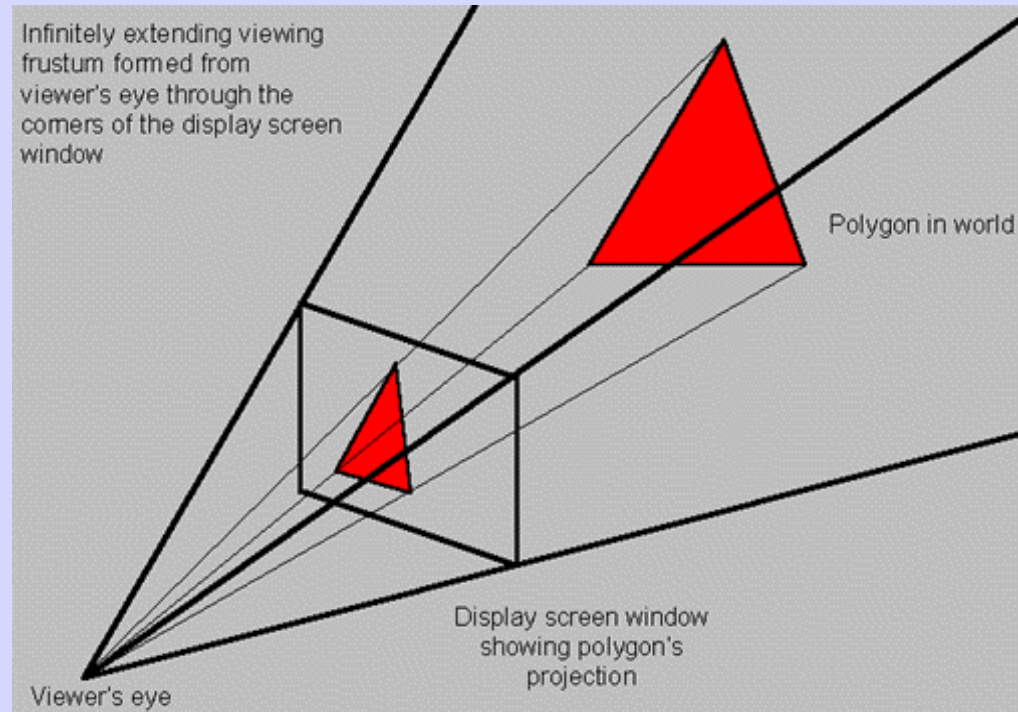   - Projection
    - E.g., color per vertex
- Clipping
- Screen Mapping

Infinitely extending viewing frustum formed from viewer's eye through the corners of the display screen window

Polygon in world

Display screen window showing polygon's projection

Viewer's eye

The RASTERIZER stage

- Main task: take output from GEOMETRY and turn into visible pixels on screen
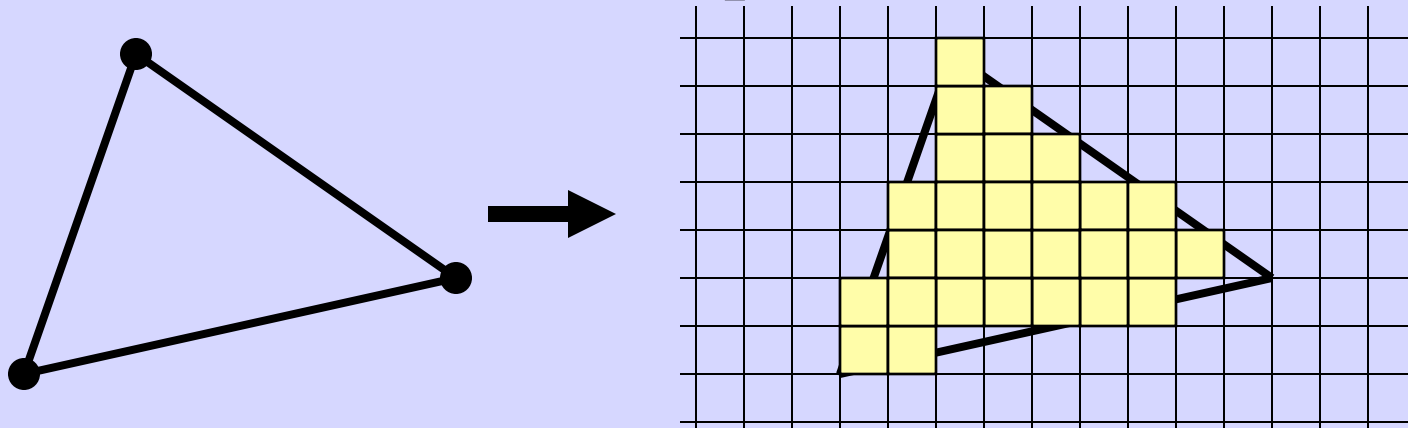
- Computes color per pixel, using fragment shader (=pixel shader)

  - textures, (light sources, normal), colors and various other per-pixel operations

- And visibility is resolved here: sorts the primitives in the z-direction

The rasterizer stage
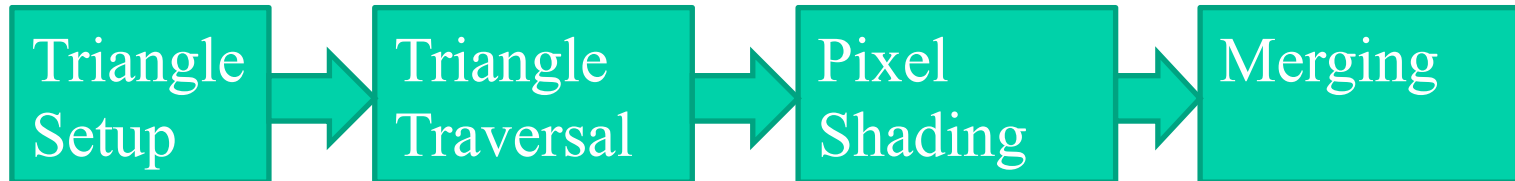
Triangle Setup	→	Triangle Traversal	→	Pixel Shading	→	Merging

Triangle Setup:
- collect three vertices + vertex shader output (incl. normals) and make one triangle.

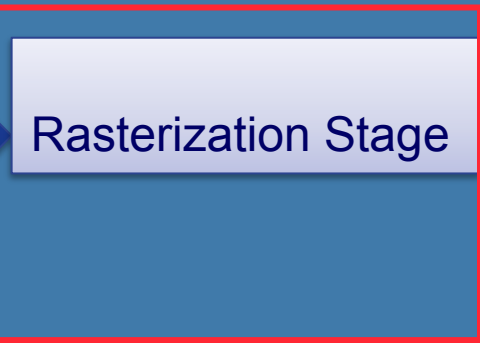Triangle Traversal
- Scan conversion

Pixel Shading
- Compute pixel color

Merging:
- output color to screen

Rendering Pipeline and Hardware

CPU

GPU

Application Stage → Geometry Stage → Rasterization Stage

Rendering Pipeline and Hardware

CPU

GPU

| Application Stage | Geometry Stage | Rasterization Stage |

HARDWARE

| Vertex shader | Geometry shader | Clipping | Screen mapping | Triangle Setup | Triangle Traversal | Pixel shader | Merger |

Display

Hardware design

Vertex shader:

• Lighting (colors)

• Screen space positions

Infinitely extending viewing frustum formed from viewer's eye through the corners of the display screen window

Polygon in world

Display screen window showing polygon's projection

Viewer's eye

• light

Geometry

blue

red   green

HARDWARE

Vertex shader → Geometry shader → Clipping → Screen mapping → Triangle Setup → Triangle Traversal → Pixel shader → Merger →
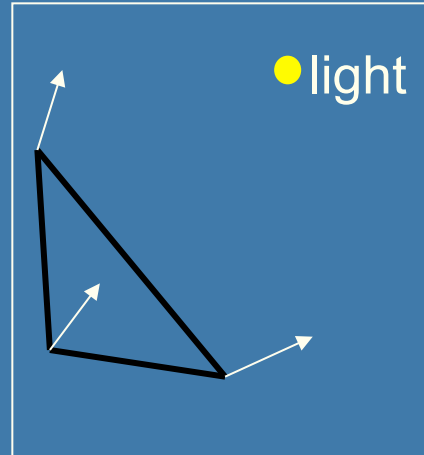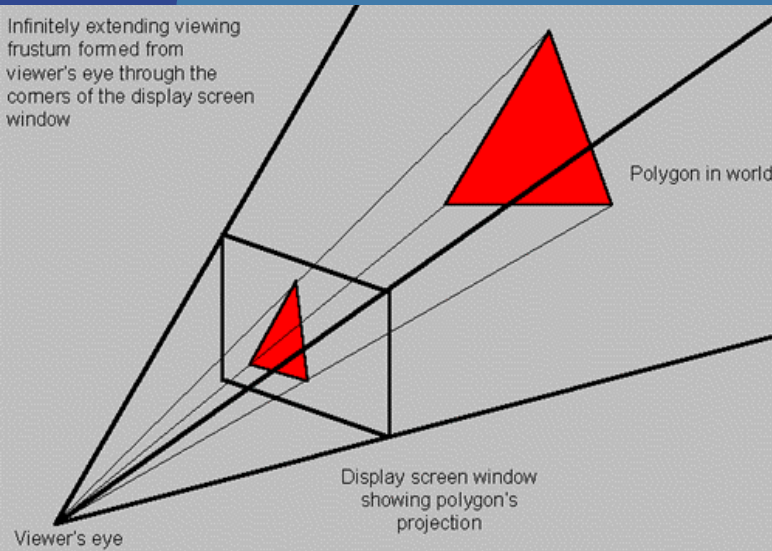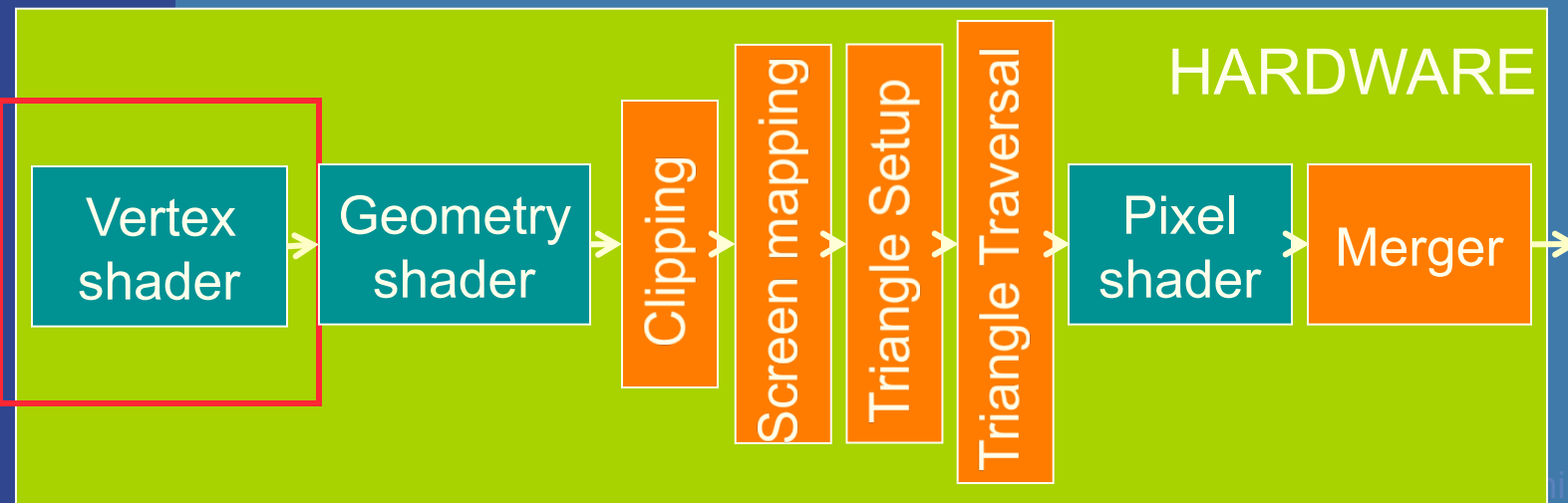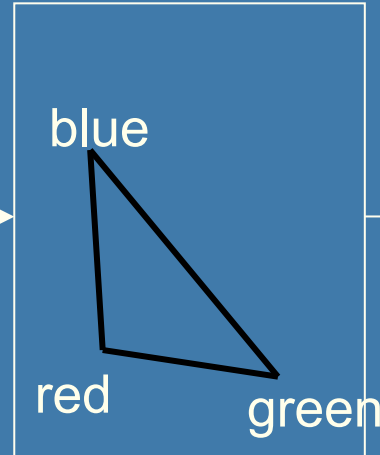
Display

Hardware design

Geometry shader:

- One input primitive

- Many output primitives

or

HARDWARE

| Vertex shader | Geometry shader | Clipping | Screen mapping | Triangle Setup | Triangle Traversal | Pixel shader | Merger |

Display

Tomas Akenine-Möller © 2003

Hardware design

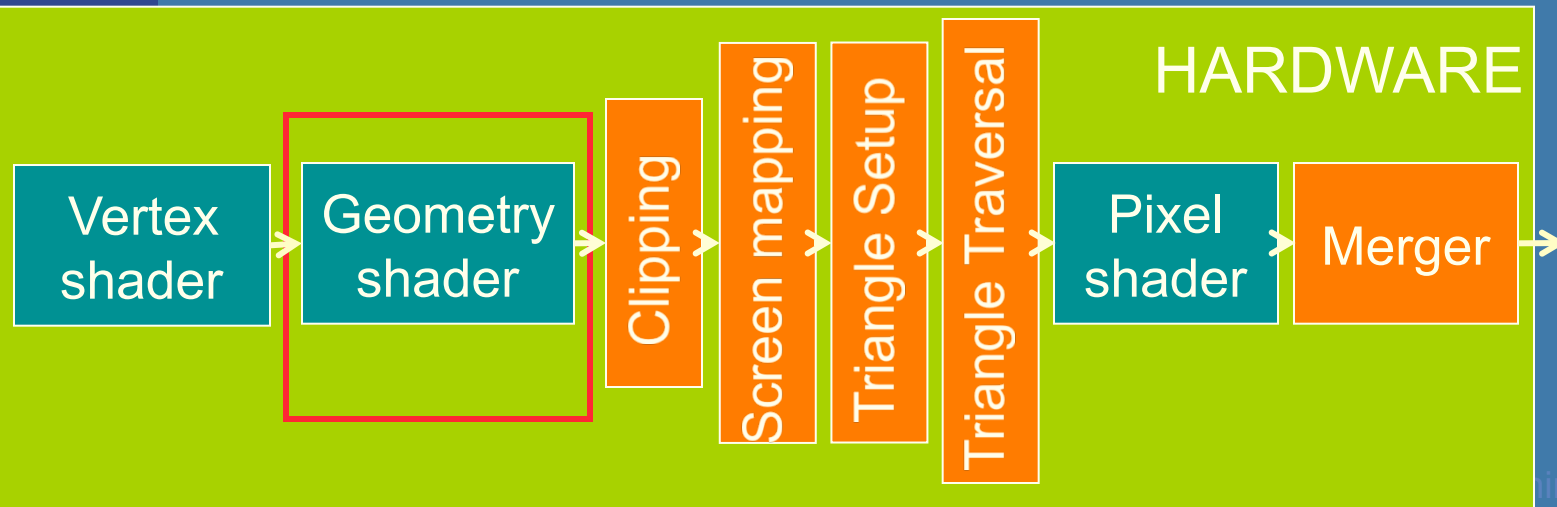Clips triangles against the unit cube (i.e., "screen borders")

HARDWARE

| Vertex shader | Geometry shader | Clipping | Screen mapping | Triangle Setup | Triangle Traversal | Pixel shader | Merger |

Display

Hardware design

Maps window size to unit cube

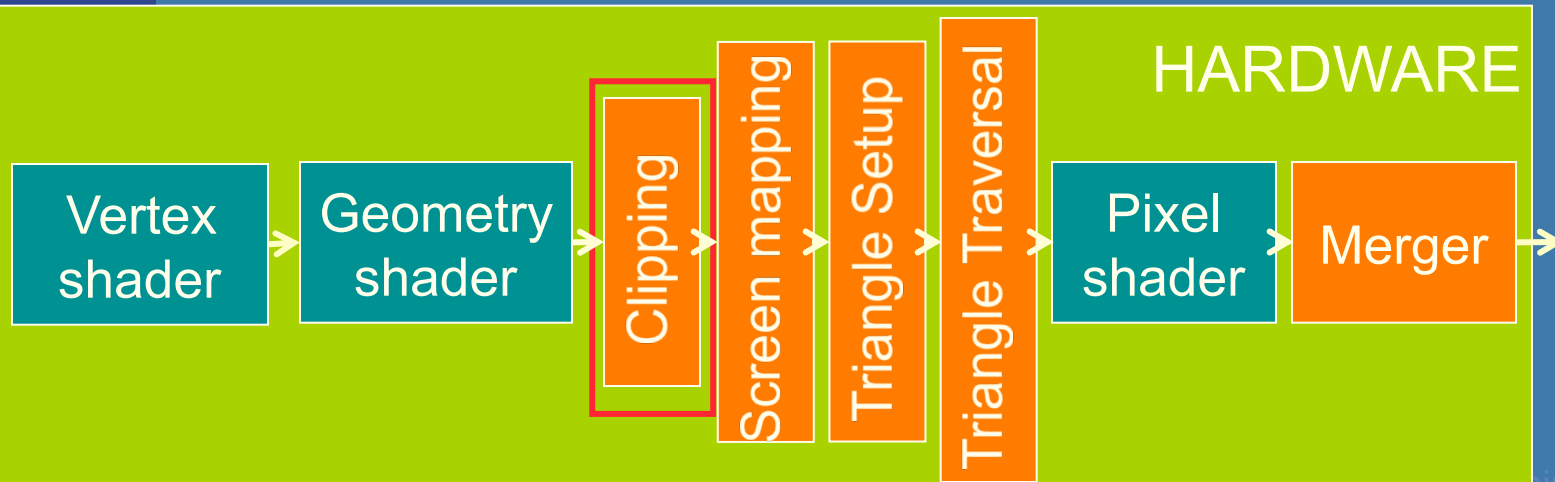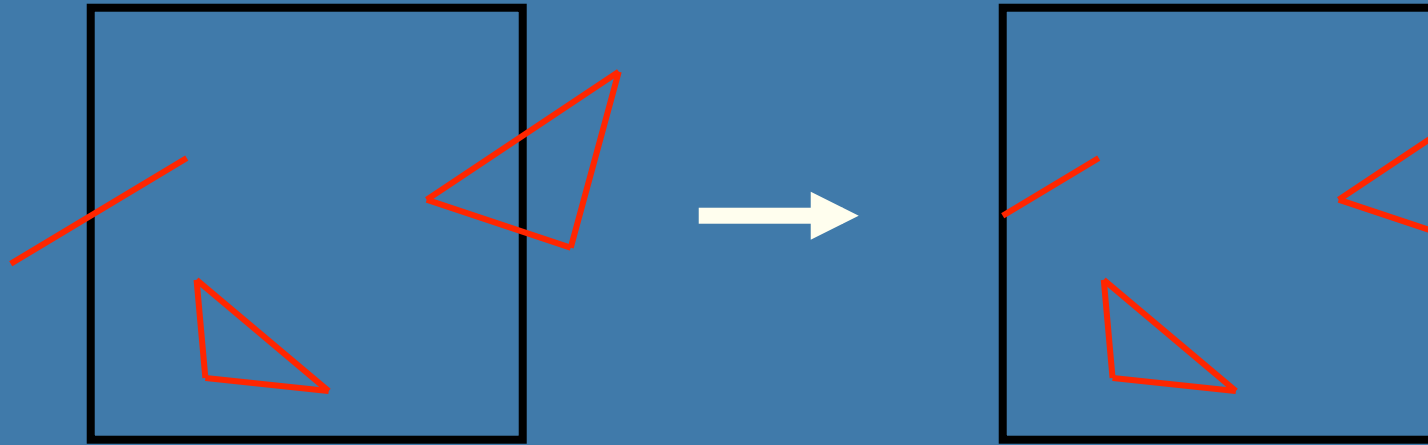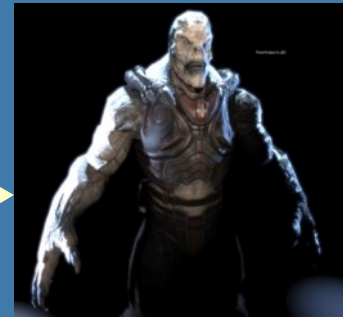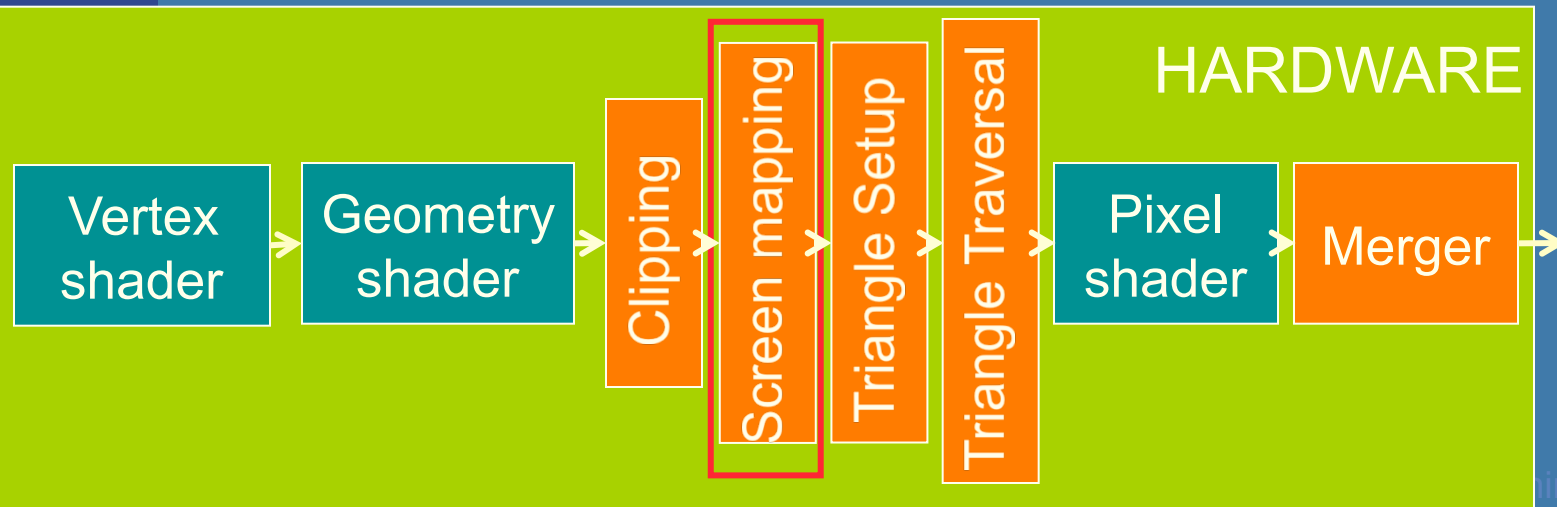Geometry stage always operates inside a unit cube [-1,-1,-1]-[1,1,1]
Next, the rasterization is made against a draw area corresponding to window dimensions.

HARDWARE

| Vertex shader | → | Geometry shader | → | Clipping | → | Screen mapping | → | Triangle Setup | → | Triangle Traversal | → | Pixel shader | → | Merger | → |
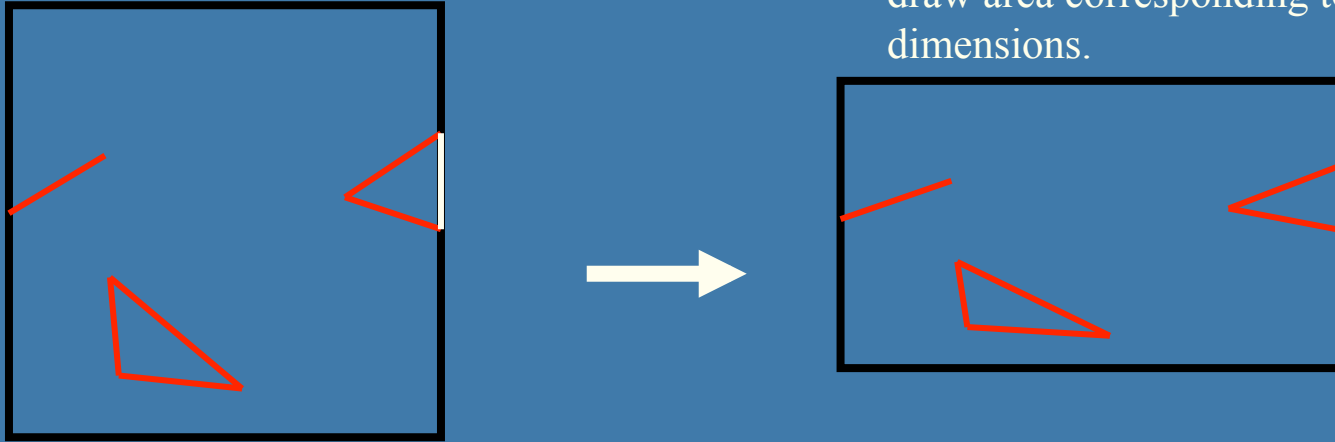
Display

Hardware design

Collects three vertices into one triangle

HARDWARE

| Vertex shader | Geometry shader | Clipping | Screen mapping | Triangle Setup | Triangle Traversal | Pixel shader | Merger |

Display

Akenine-Möller © 2003
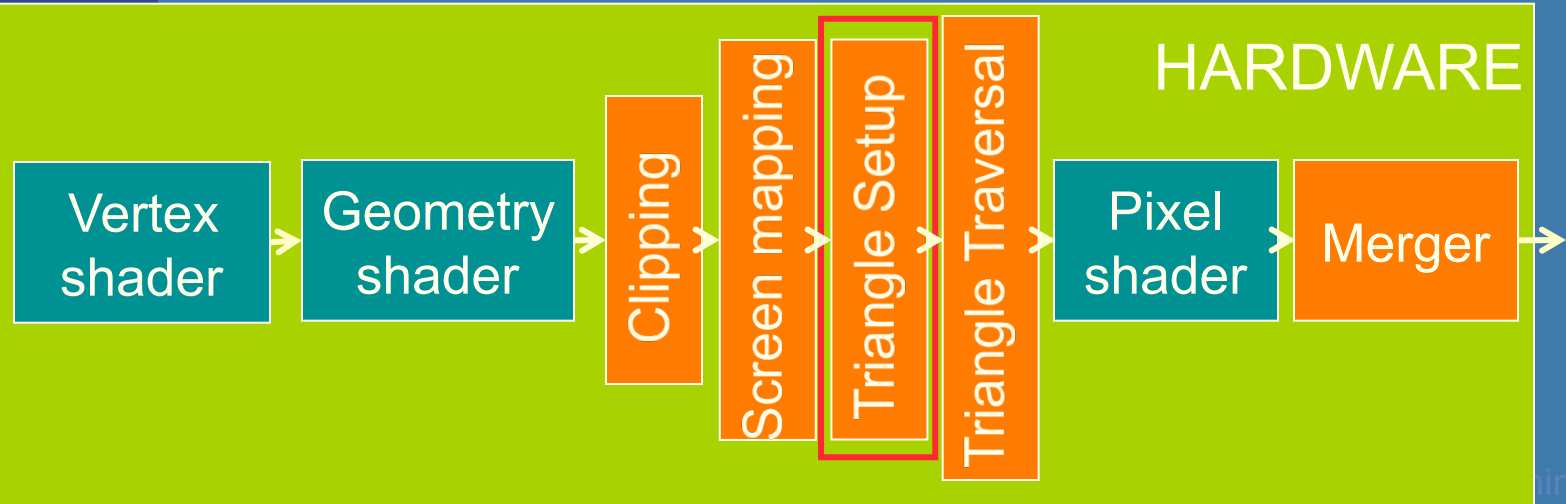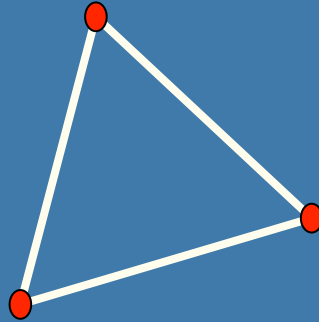
Hardware design

Creates the fragments/ pixels for the triangle
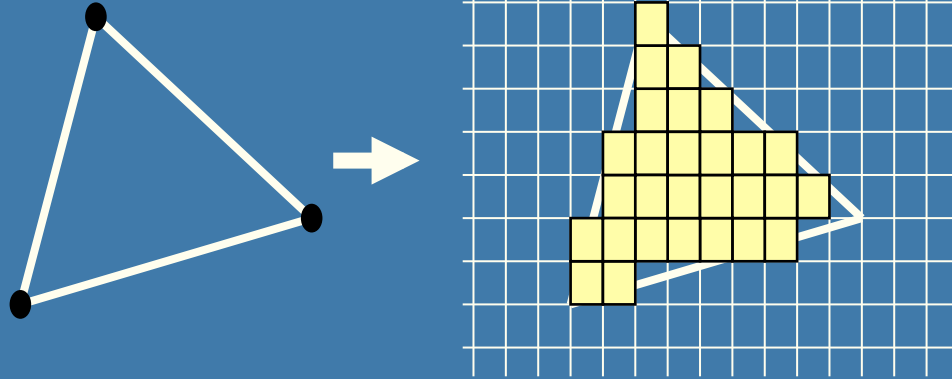
HARDWARE

| Vertex shader | Geometry shader | Clipping | Screen mapping | Triangle Setup | Triangle Traversal | Pixel shader | Merger |

Display

Hardware design

blue

red

green
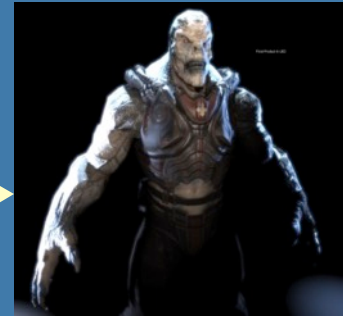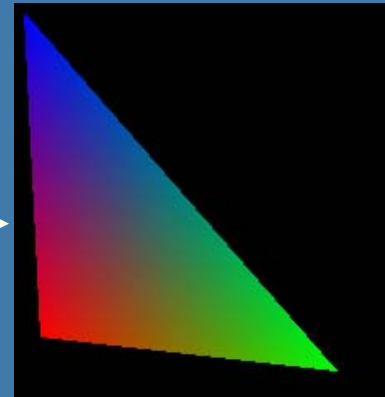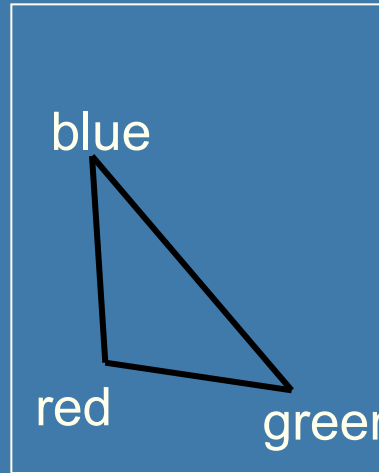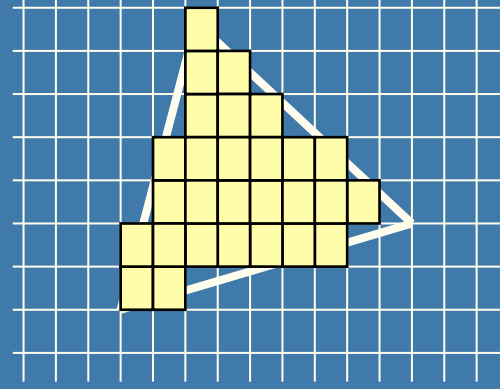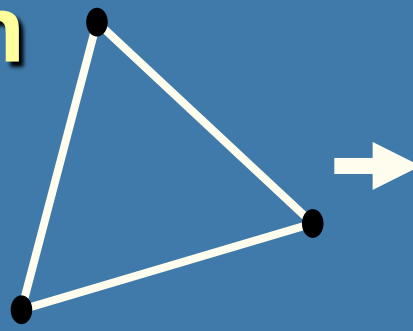
Rasterizer

Pixel Shader:
Compute color
using:
•Textures
•Interpolated data
(e.g. Colors +
normals) from
vertex shader

HARDWARE

| Vertex shader | Geometry shader | Clipping | Screen mapping | Triangle Setup | Triangle Traversal | Pixel shader | Merger |
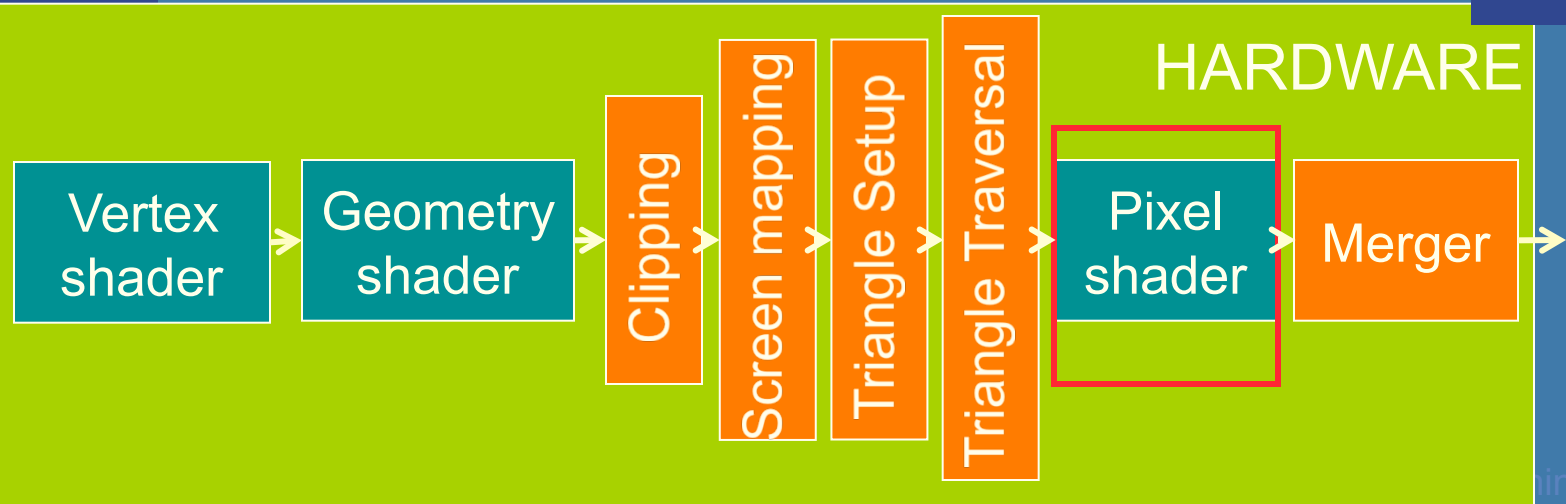
Display

Hardware design

The merge units update the frame buffer with the pixel's color

Frame buffer:

- Color buffers

- Depth buffer

- Stencil buffer

HARDWARE

| Vertex shader	Geometry shader	Clipping	Screen mapping	Triangle Setup	Triangle Traversal	Pixel shader	Merger
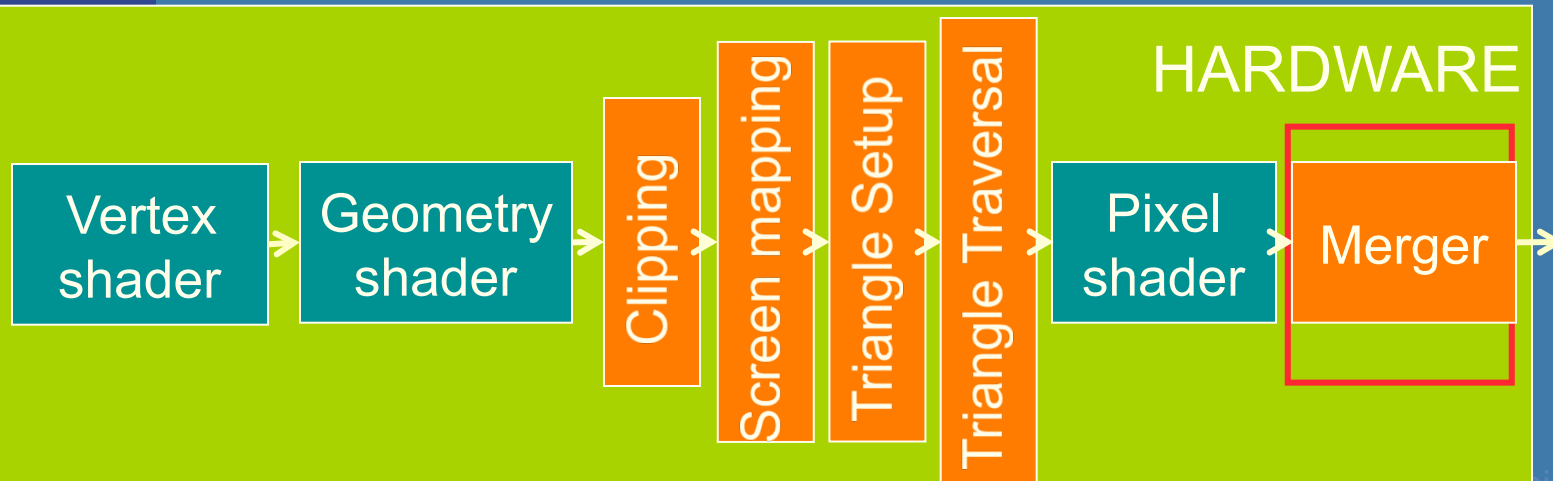
Display

What is vertex and fragment (pixel) shaders?
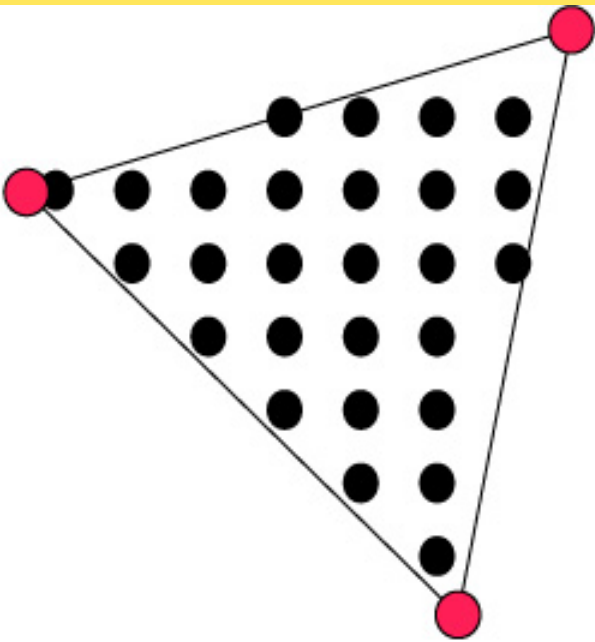
- Vertex shader: reads from textures
- Fragment shader: reads from textures, writes to pixel color
- Memory: Texture memory (read + write) typically 500 Mb – 4 GB
- Program size: the smaller the faster
- Instructions: mul, rcp, mov, dp, rsq, exp, log, cmp, jnz…

For each vertex, a vertex program (vertex shader) is executed

For each fragment (pixel) a fragment program (fragment shader) is executed

Shaders
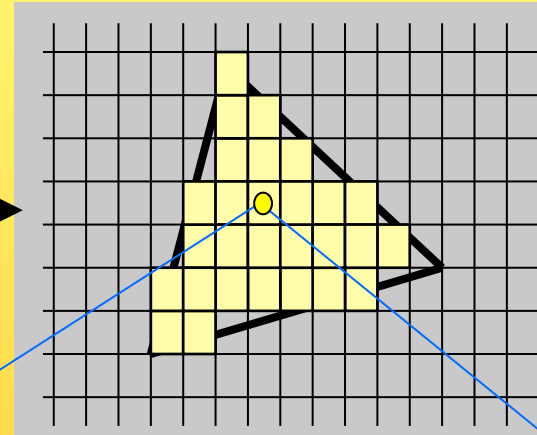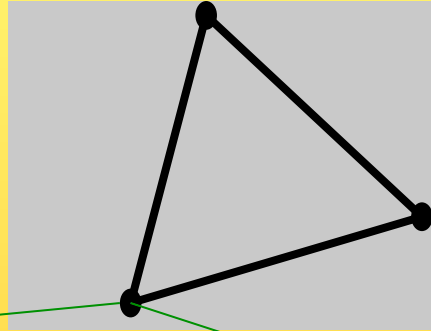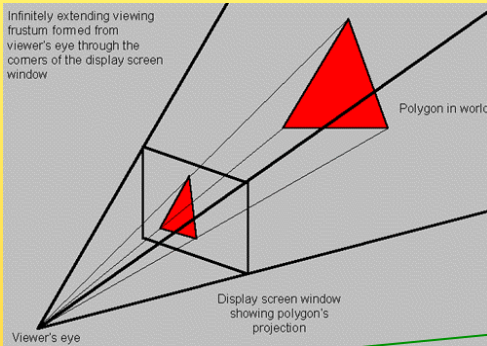

```glsl
// Vertex Shader
#version 130

in  vec3 vertex;
in  vec3 color;
out vec3 outColor;
uniform mat4 modelViewProjectionMatrix;

void main()
{
    gl_Position = modelViewProjectionMatrix*vec4(vertex,1);
    outColor = color;
}
```
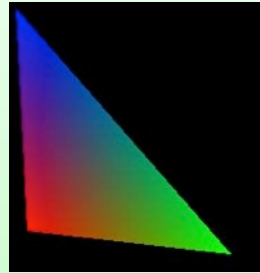


```glsl
// Fragment Shader:
#version 130

in  vec3 outColor;
out vec4 fragColor;

void main()
{
    fragColor = vec4(outColor,1);
}
```

Fragment Shader

```
precision highp float;

uniform sampler2D tex0;
uniform sampler2D tex1;
uniform sampler2D tex2;
uniform sampler2D tex3;

uniform float val;

varying vec2 uv_0;
varying vec3 n;

void main(void) {
    gl_FragColor.rgb = compute_color();
    gl_FragColor.a = 1.0;
}
```

```
vec3 compute_color()
{
    vec4 gbuffer = texture2D(tex0, uv_0);
        int intColor = int(gbuffer.x);
        int r = (intColor/256)/256;
        intColor -= r*256*256;
        int g = intColor/256;
        intColor -= g*256;
        int b = intColor;
         vec3 color = vec3(float(r)/255.0, float(g)/255.0,
        float(b)/255.0 );

        normal = vec3(sin(gbuffer.g) * cos(gbuffer.b),
        sin(gbuffer.g)*sin(gbuffer.b), cos(gbuffer.g));
        vec2 ang = gbuffer.gb*2.0-vec2(1.0);
        vec2 scth = vec2( sin(ang.x * PI), cos(ang.x * PI);
        vec2 scphi = vec2(sqrt(1.0 - ang.y*ang.y), ang.y);
        normal = -vec3(scth.y*scphi.x, scth.x*scphi.x, scphi.y);
        roughness = 0.05;
        specularity = 1.0;
        fresnelR0 = 0.3;
        return color;
}
```
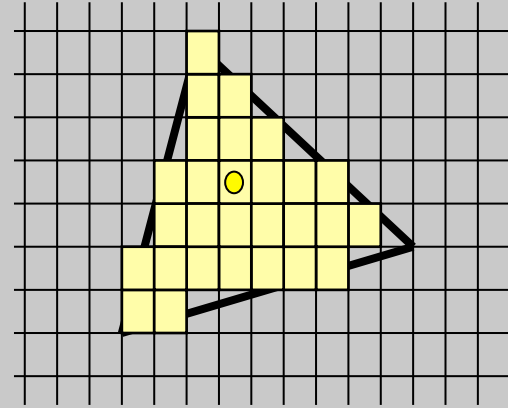
Cg - "C for Graphics" (NVIDIA)

```
if (slice >= 0.0h) {
    half gradedEta = BallData.ETA;
    gradedEta = 1.0h/gradedEta;   // test hack
    half3 faceColor = BgColor; // blown out - go to BG color
    half c1 = dot(-Vn,Nf);
    half cs2 = 1.0h-gradedEta*gradedEta*(1.0h-c1*c1);

    if (cs2 >= 0.0h) {
        half3 refVector = gradedEta*Vn+((gradedEta*c1-sqrt(cs2))*Nf);
        // now let's intersect with the iris plane
        half irisT = intersect_plane(IN.OPosition,refVector,planeEquation);
        half fadeT = irisT * BallData.LENS_DENSITY;
        fadeT = fadeT * fadeT;
        faceColor = DiffPupil.xxx;   // temporary (?)
        if (irisT > 0) {
            half3 irisPoint = IN.OPosition + irisT*refVector;
            half3 irisST = (irisScale*irisPoint) + half3(0.0h,0.5h,0.5h);
            faceColor = tex2D(ColorMap,irisST.yz).rgb;
        }
        faceColor = lerp(faceColor,LensColor,fadeT);
        hitColor = lerp(missColor,faceColor,smoothstep(0.0h,GRADE,slice));
    }
}
```

PixelShader 3.0

```
// if (-dir.z/|dir| > cos(PI/4)) t1 = zero
dp3 r6.w, r6, r6          ←normalization
rsq r6.w, r6.w
mad r0.w, -r6.z, r6.w, -CosPiOverFour
cmp r10.y, r0.w, Zero, r10.y

// set r10 to 0 if Disc <= 0
cmp r10.xy, -r7.w, Zero, r10

// compute r1 and r2 clipped
mad r1.xyz, r6, r10.x, r4              // IP0
mad r2.xyz, r6, r10.y, r4              // IP1

// project
rcp r11.w, r1.z
mad r1.xyz, r1, r11.w, NegZ            // P0
rcp r11.w, r2.z
mad r2.xyz, r2, r11.w, NegZ            // P1

// Compute area
texld r3, r1, ATan2Texture            // theta0
texld r4, r2, ATan2Texture            // theta1

crs r5.z,r1,r2                                // z = z
abs r5.z,r5.z

mov r3.y, r4.x
texld r4, r3, SphAreaTexture          // lookup theta/PI
```
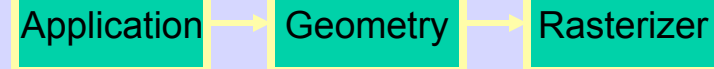
- Float, int
- Instructions operate on 1,2,3 or 4 components
  - x,y,z,w or
  - r,g,b,a
- Free Swizzling
- Only read from texture
- (Only write to pixel (8 output buffers))
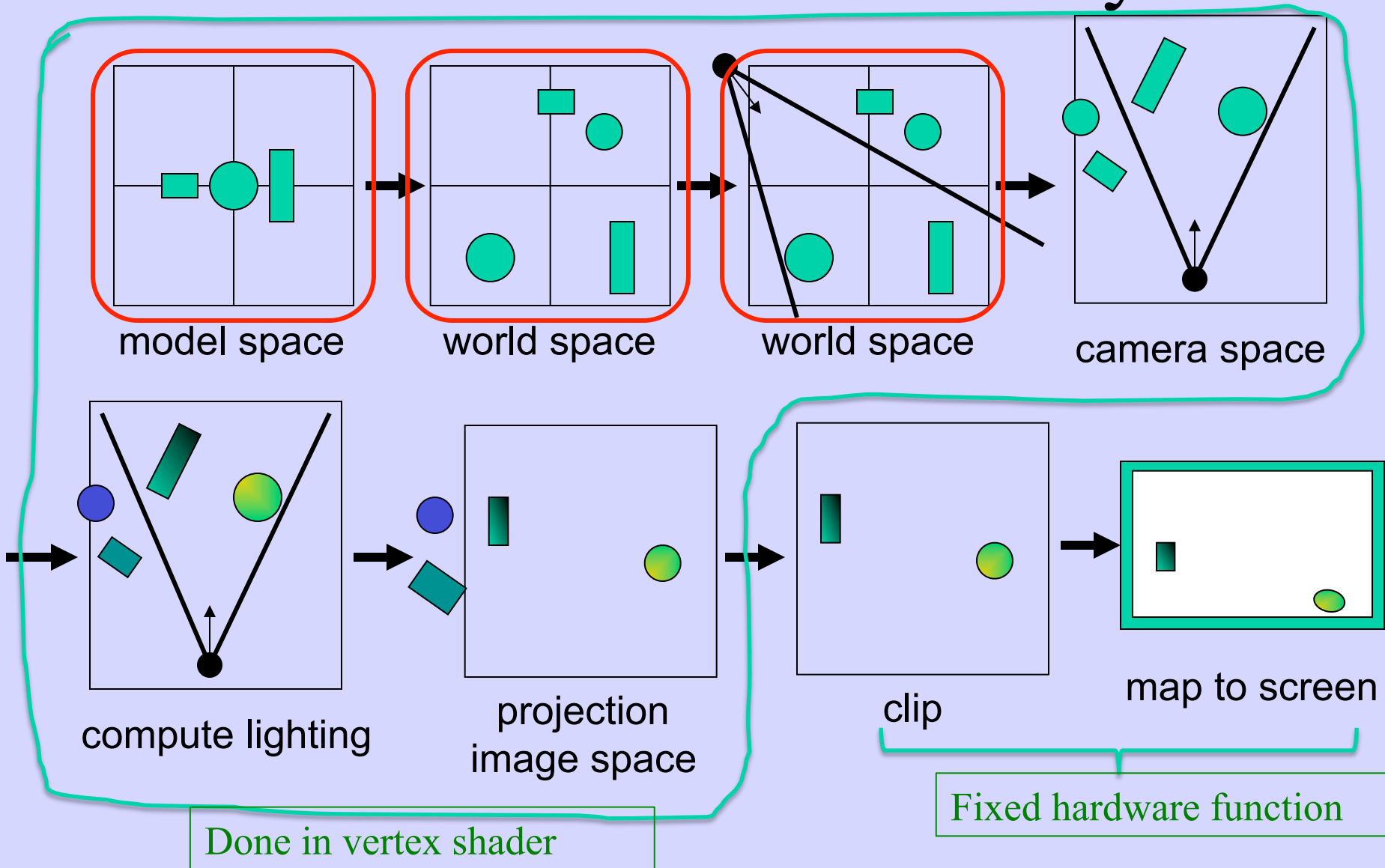
Rewind!
Let's take a closer look

- 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
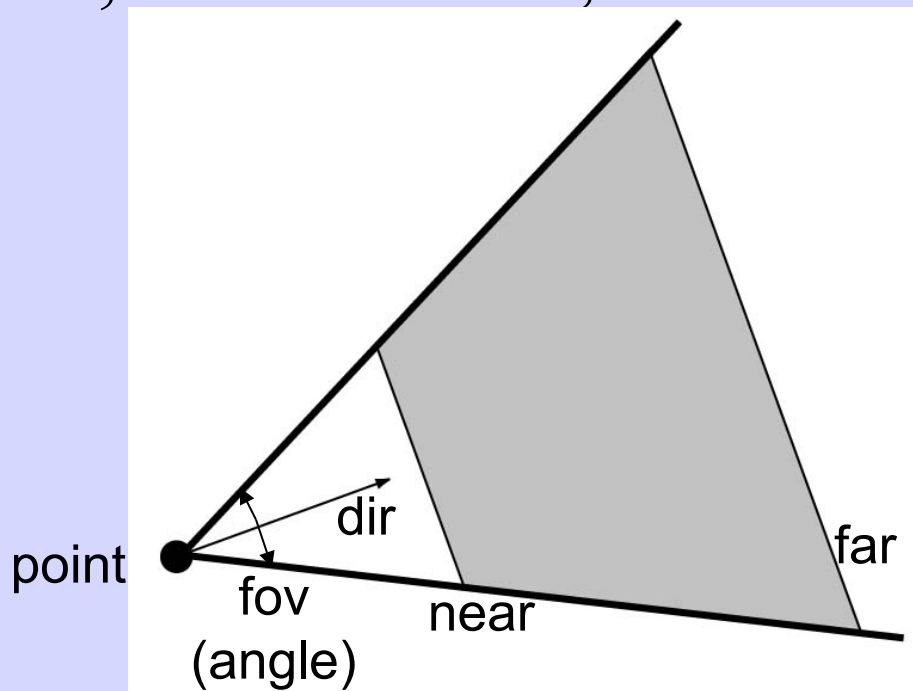
Application → Geometry → Rasterizer

model space

world space

world space

camera space

compute lighting

projection
image space

clip

map to screen

Done in vertex shader

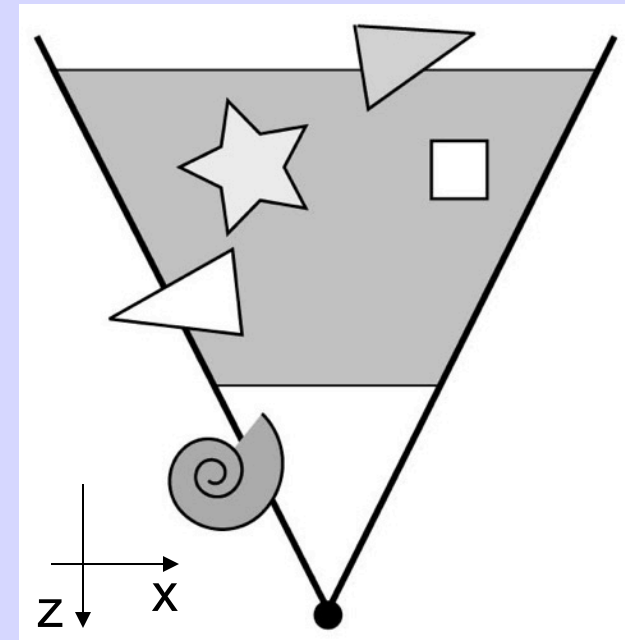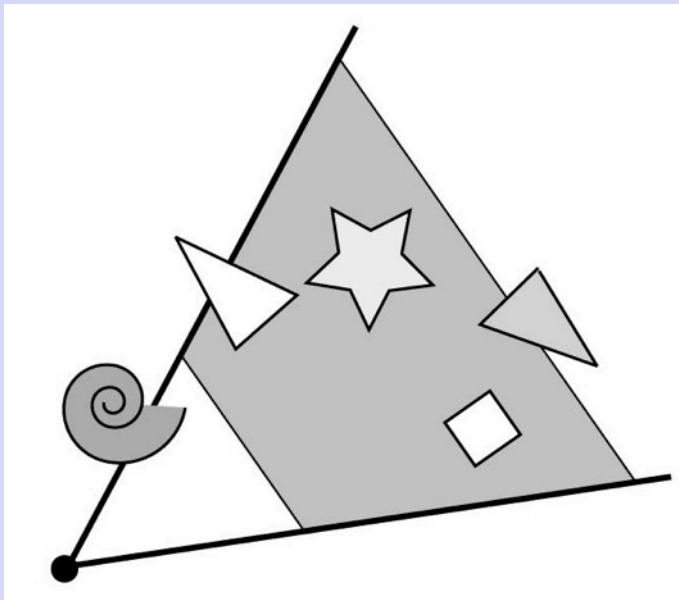Fixed hardware function

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
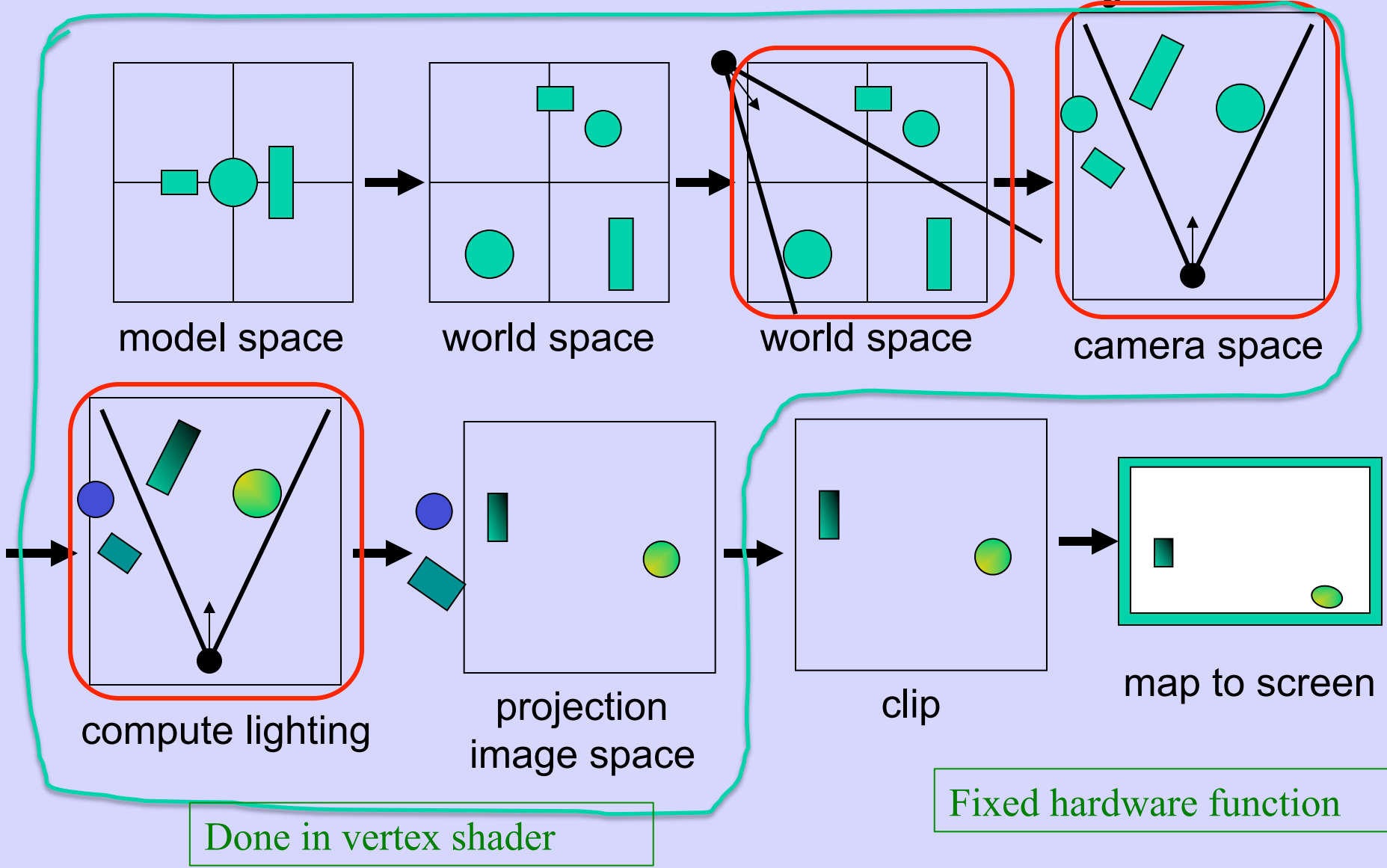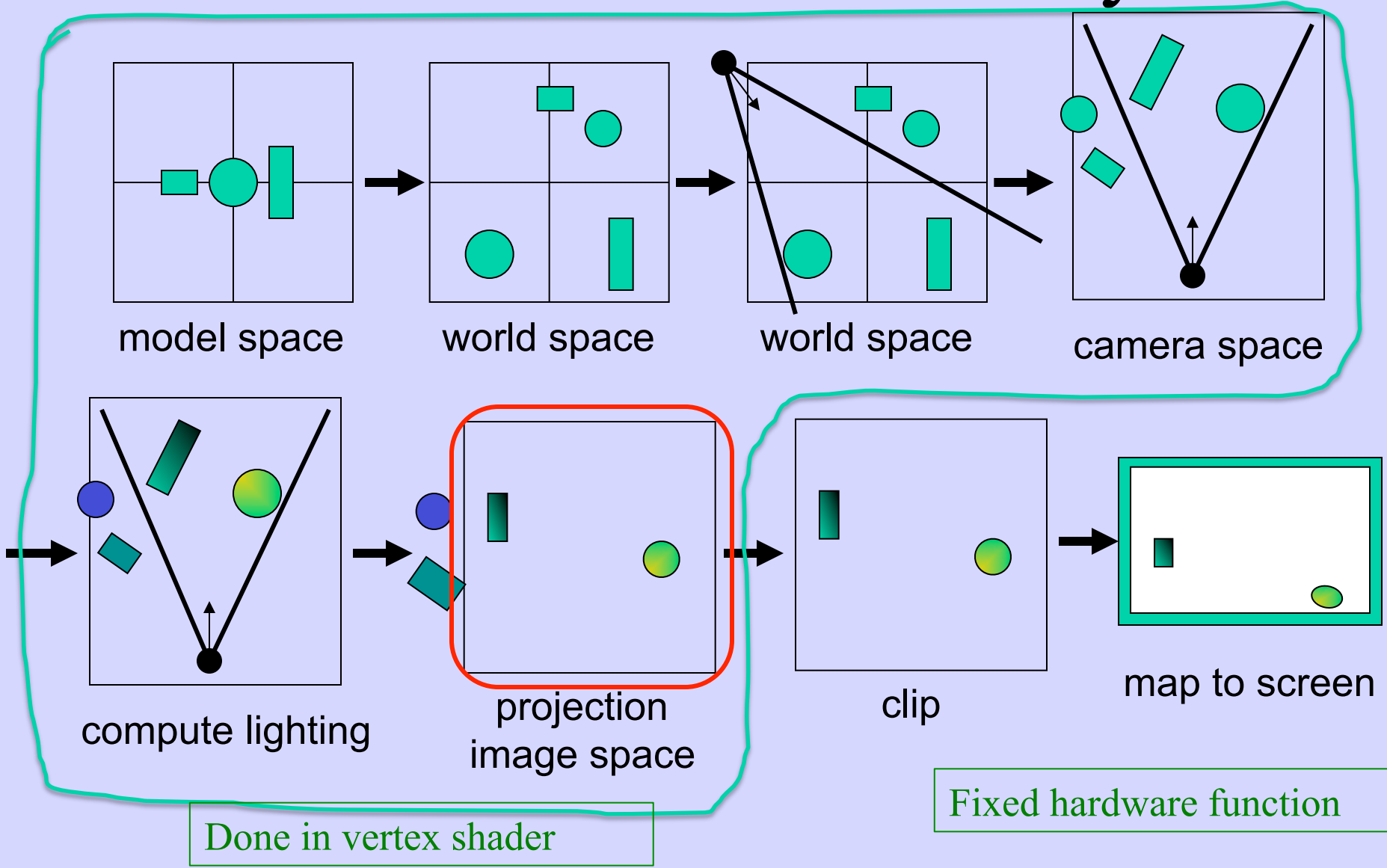

z ↓ x →

GEOMETRY - Summary

Application → Geometry → Rasterizer

model space

world space

world space

camera space

compute lighting

projection
image space

clip

map to screen

Done in vertex shader

Fixed hardware function
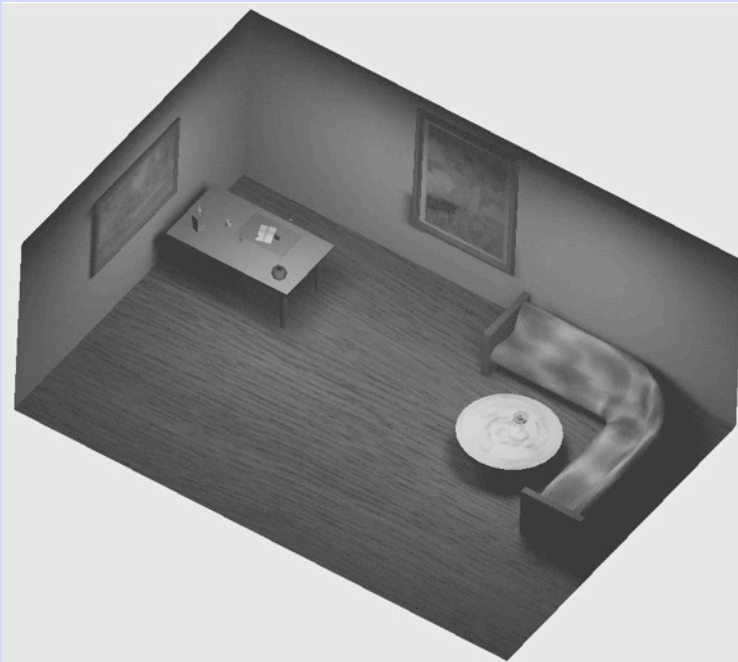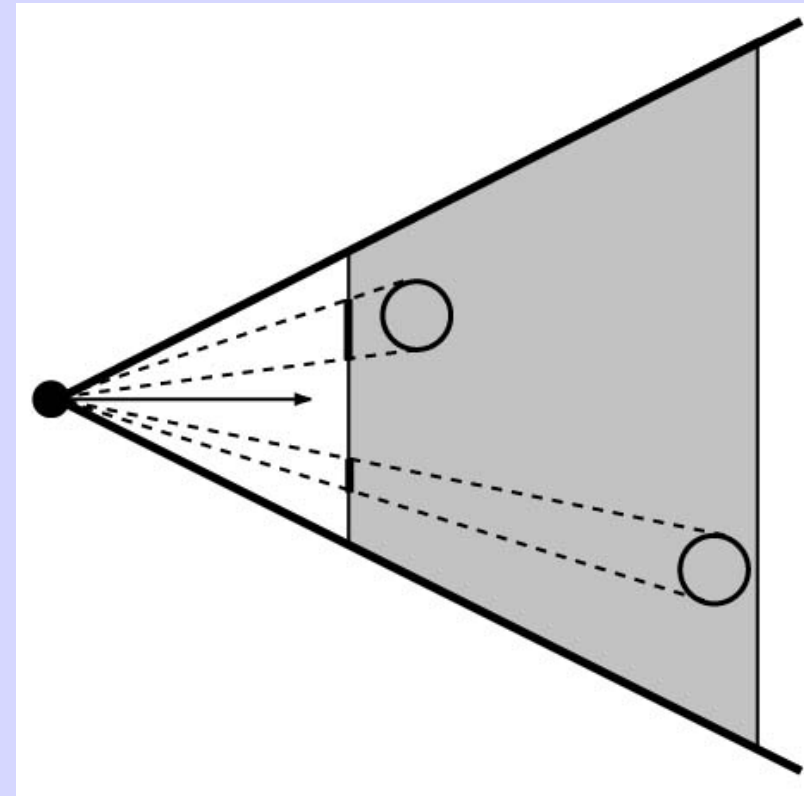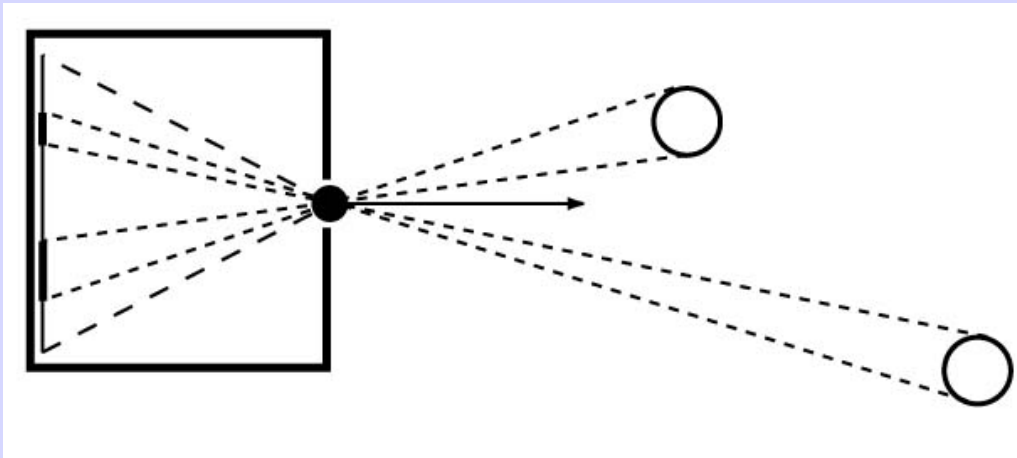
GEOMETRY - Projection

- 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
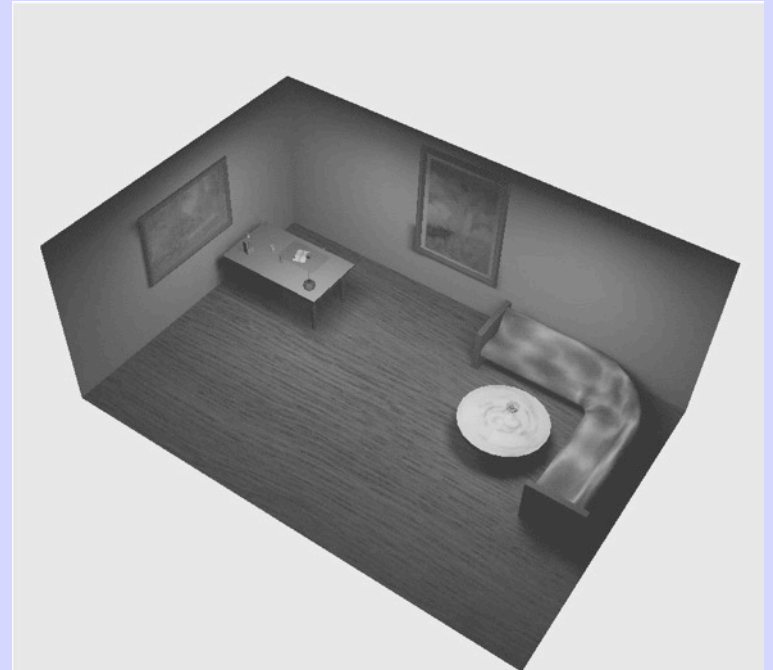
GEOMETRY - Projection

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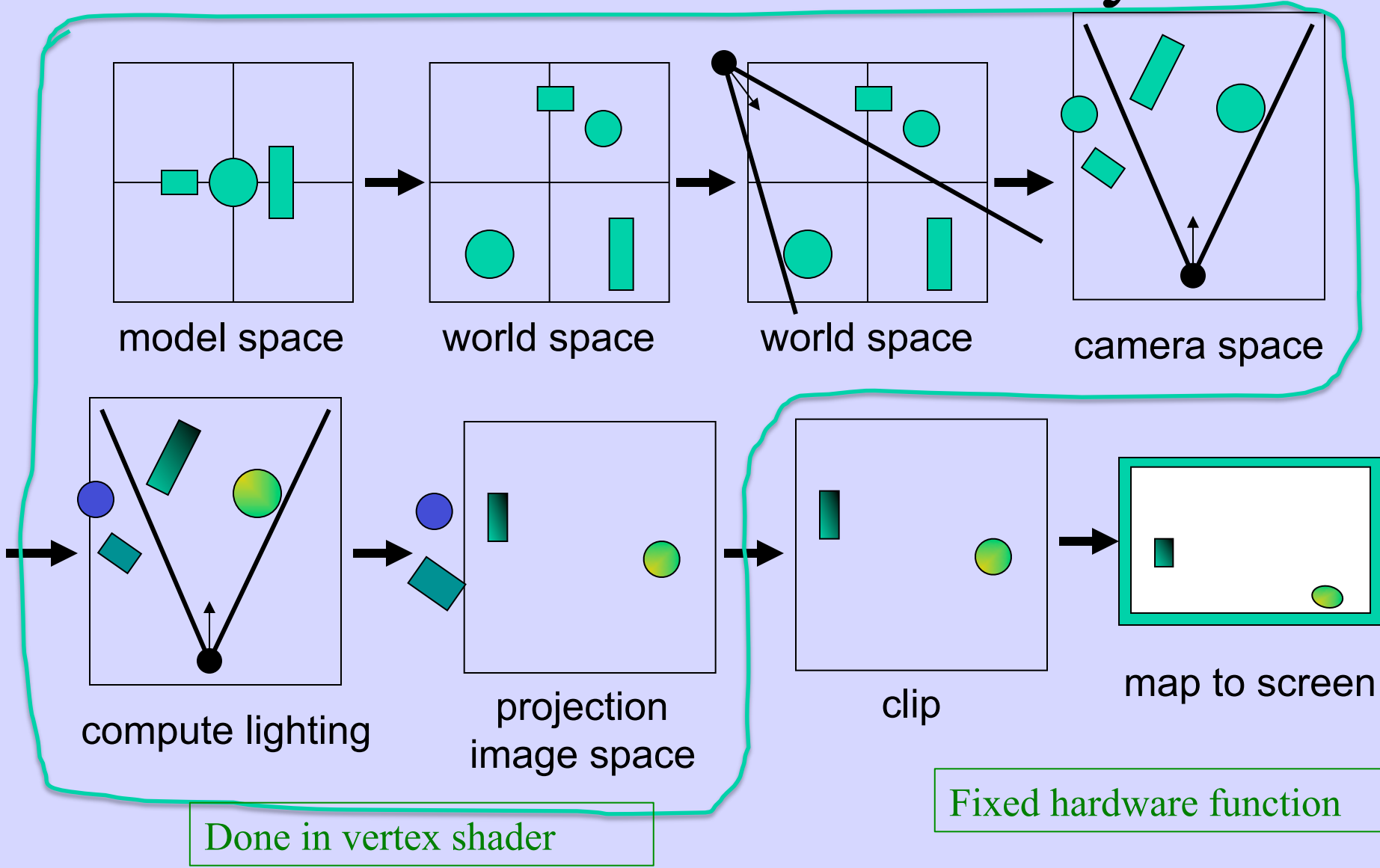- Pinhole camera (left), analog used in CG (right)

GEOMETRY - Summary

model space

world space

world space

camera space

compute lighting

projection
image space

clip

map to screen

Done in vertex shader

Fixed hardware function
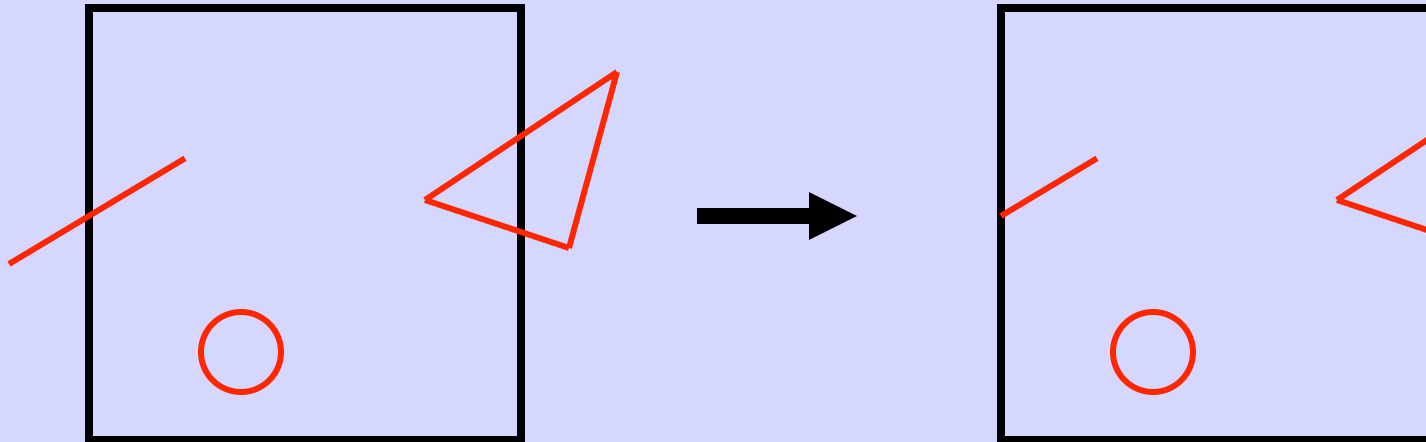
GEOMETRY
Clipping and Screen Mapping

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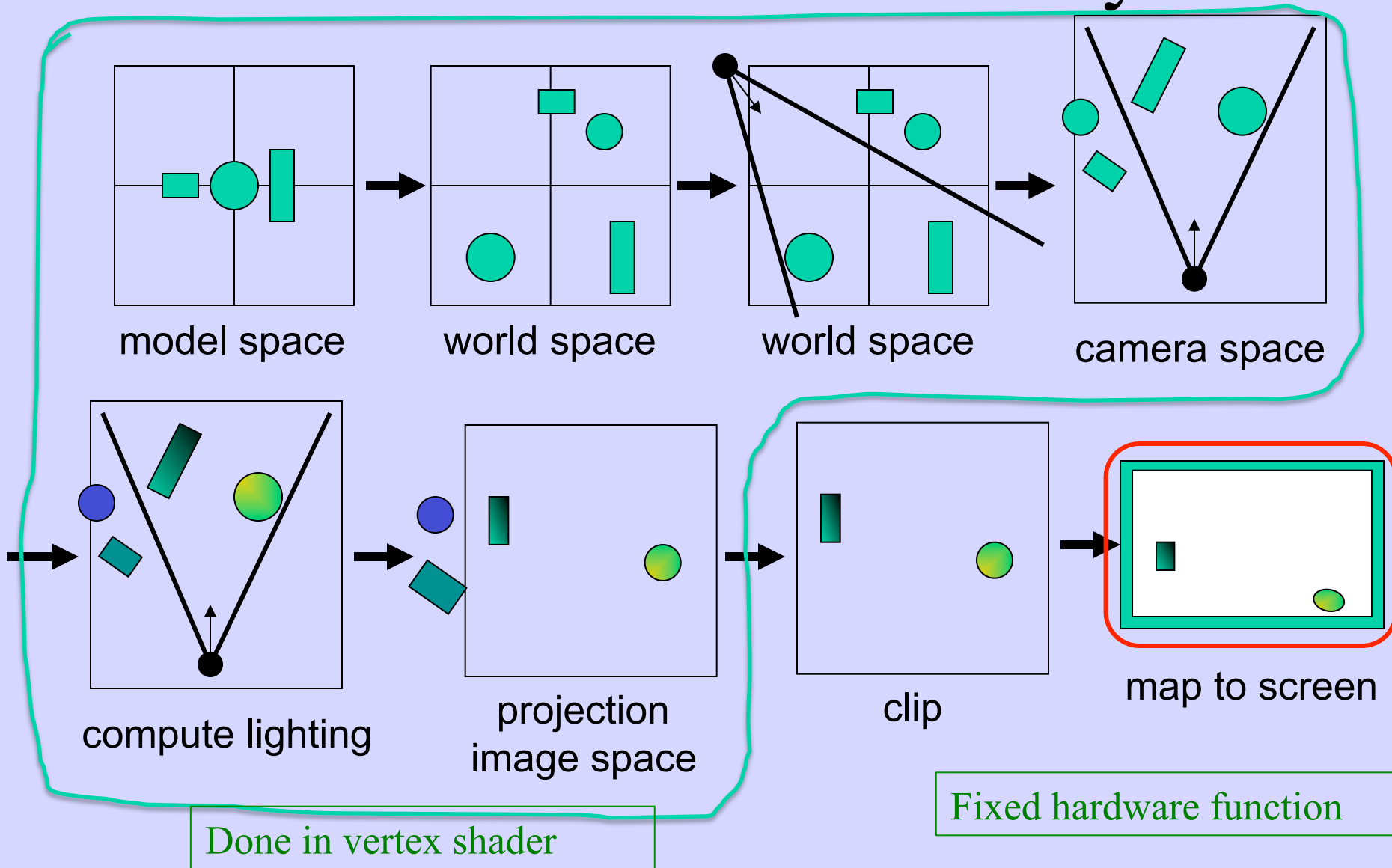- Clip primitives to square

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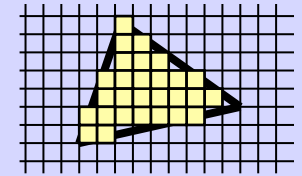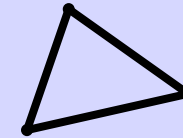
Application → Geometry → Rasterizer

GEOMETRY - Summary

model space

world space

world space

camera space

compute lighting

projection
image space

clip

map to screen

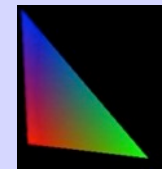Done in vertex shader

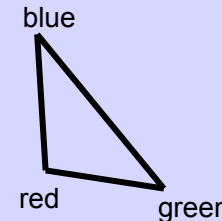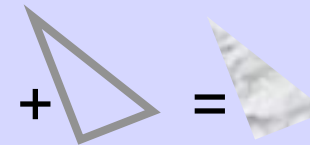Fixed hardware function

The RASTERIZER in more detail

- ## Scan-conversion
  - Find out which pixels are inside the primitive

- ## Fragment shaders
  - E.g. put textures on triangles
  - Use interpolated data over triangle
  - and/or compute per-pixel lighting

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

- ## Doublebuffering
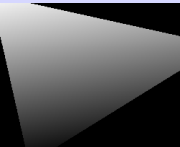
The RASTERIZER
Z-buffering

- A triangle that is covered by a more closely located triangle should not be visible

- Assume two equally large tris at different depths

incorrect   correct
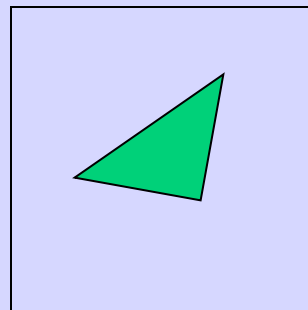
Triangle 1   Triangle 2   Draw 1 then 2   Draw 2 then 1
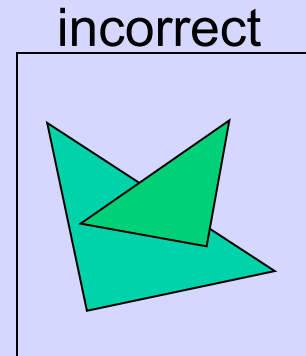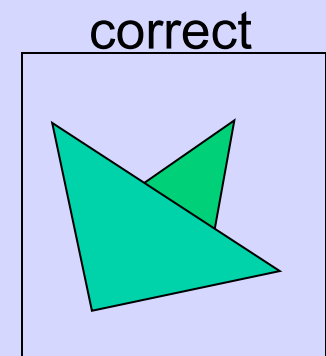
The RASTERIZER
Z-buffering
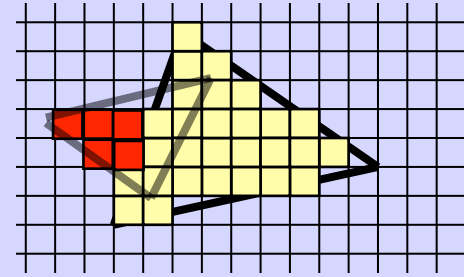
- Would be nice to avoid sorting…
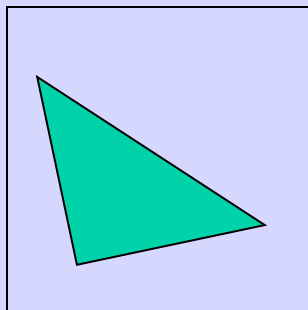- The Z-buffer (aka depth buffer) solves this
- Idea:
  - Store z (depth) at each pixel
  - When rasterizing a triangle, compute z at each pixel on triangle
  - Compare triangle's z to Z-buffer z-value
  - If triangle's z is smaller, then replace Z-buffer and color buffer
  - Else do nothing
- Can render in any order

Painter's Algorithm

- Render polygons a back to front order so that polygons behind others are simply painted over
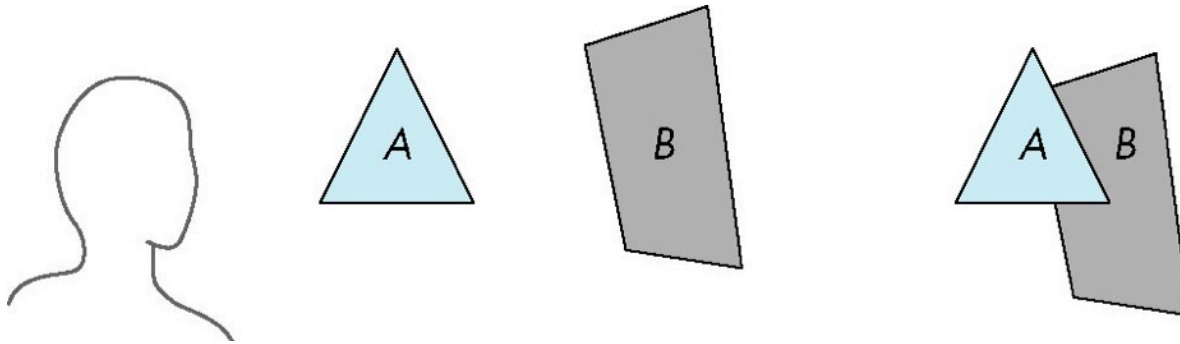
B behind A as seen by viewer

Fill B then A

- Requires ordering of polygons first
  - O(n log n) calculation for ordering
  - Not every polygon is either in front or behind all other polygons

I.e., : Sort all triangles and render them back-to-front.

z-Buffer Algorithm

- Use a buffer called the z or depth buffer to store the depth of the closest object at each pixel found so far

- As we render each polygon, compare the depth of each pixel to depth in z buffer

- If less, place shade of pixel in color buffer and update z buffer
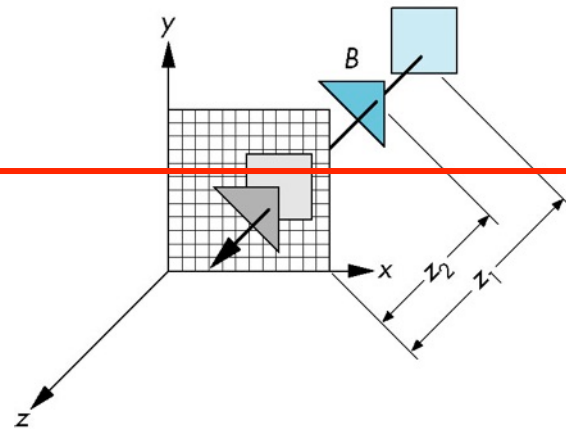
The RASTERIZER double-buffering
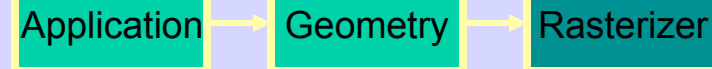
- The monitor displays one image at a time
- Top of screen – new image

  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).
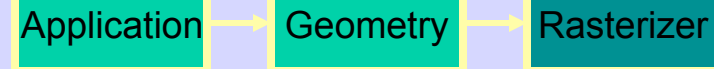
The RASTERIZER double-buffering

- 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

Screen Tearing

Swapping back/
front buffers

vblank

Screen Tearing

- Despite the gorgeous graphics seen in many of today's games, there are still some highly distracting artifacts that appear in gameplay despite our best efforts to suppress them. The most jarring of these is screen tearing. Tearing is easily observed when the mouse is panned from side to side. The result is that the screen appears to be torn between multiple frames with an intense flickering effect. Tearing tends to be aggravated when the framerate is high since a large number of frames are in flight at a given time, causing multiple bands of tearing.

- Vertical sync (V-Sync) is the traditional remedy to this problem, but as many gamers know, V-Sync isn't without its problems. The main problem with V-Sync is that when the framerate drops below the monitor's refresh rate (typically 60 fps), the framerate drops disproportionately. For example, dropping slightly below 60 fps results in the framerate dropping to 30 fps. This happens because the monitor refreshes at fixed internals (although an LCD doesn't have this limitation, the GPU must treat it as a CRT to maintain backward compatibility) and V-Sync forces the GPU to wait for the next refresh before updating the screen with a new image. This results in notable stuttering when the framerate dips below 60, even if just momentarily.
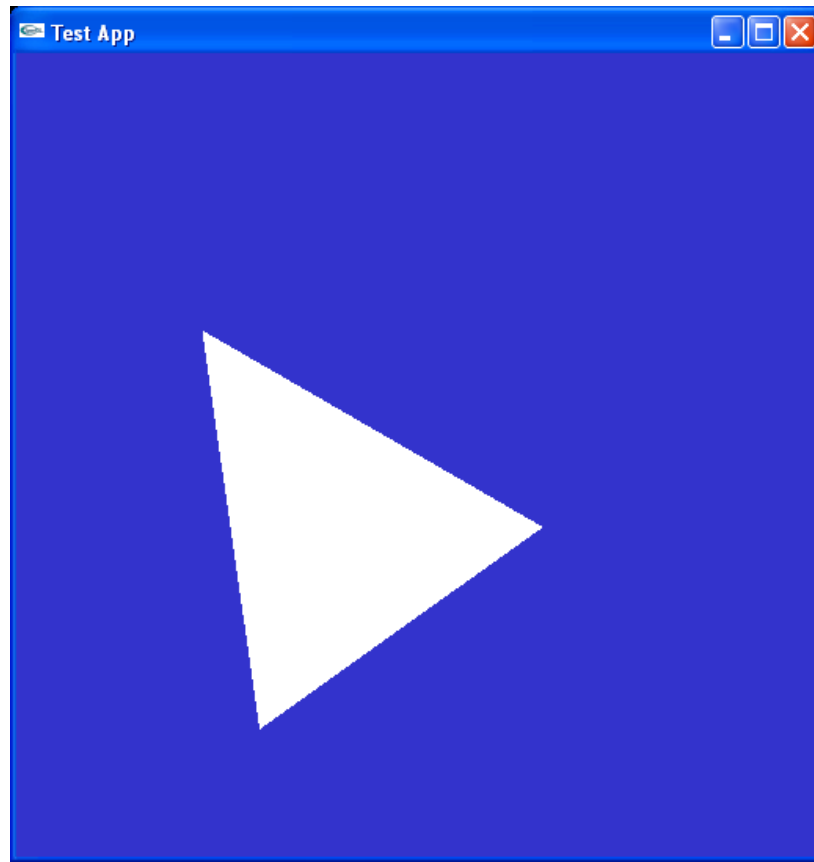
OpenGL

A Simple Program
Computer Graphics version of
"Hello World"
Generate a triangle on a solid background

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;
}
```

```cpp
void display(void)
{
    glClearColor(0.2,0.2,0.8,1.0);      // Set clear color  - for background
    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 (OpenGL draws with this resolution)

    glDisable(GL_CULL_FACE);
    drawScene();

    glutSwapBuffers();                  // swap front and back buffer. This frame will now been displayed.
}
```
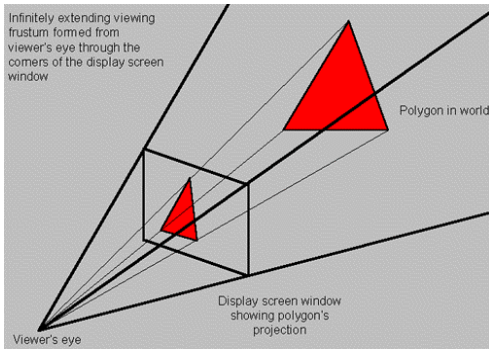
```
static void drawScene(void)
{
    // Shader Program
    glUseProgramObjectARB( shaderProgram );  // Set the shader program to use for this draw call
    CHECK_GL_ERROR();

    glBindVertexArray(vertexArrayObject);     // Tells which vertex arrays to use
    CHECK_GL_ERROR();

    glDrawArrays( GL_TRIANGLES, 0, 3 );       // Render the three first vertices as a triangle
    CHECK_GL_ERROR();
}
```

Shaders


```glsl
// Vertex Shader
#version 130

in  vec3 vertex;
in  vec3 color;
out vec3 outColor;
uniform mat4 modelViewProjectionMatrix;

void main()
{
    gl_Position = modelViewProjectionMatrix*vec4(vertex,1);
    outColor = color;
}
```
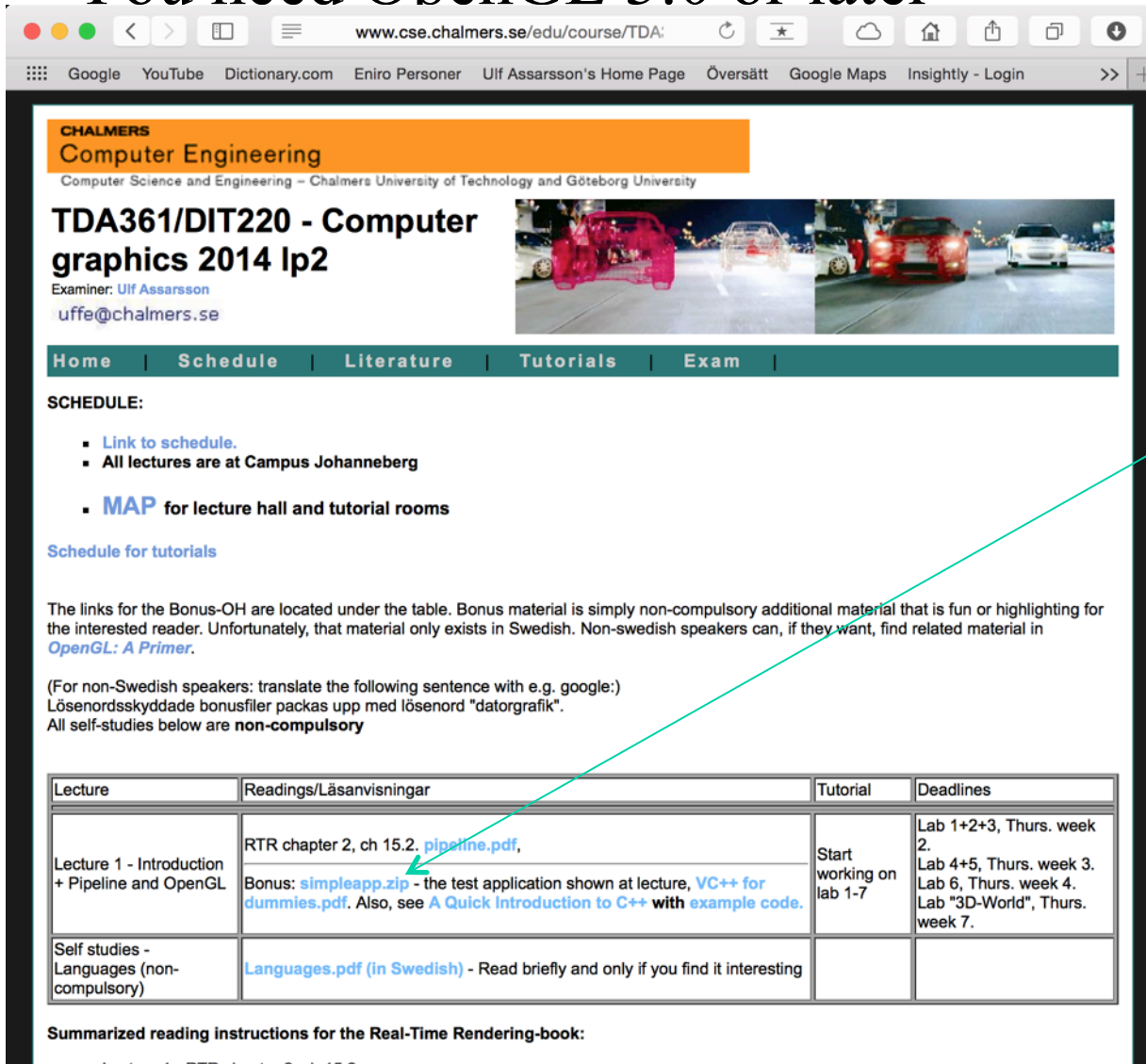
```glsl
// Fragment Shader:
#version 130
in  vec3 outColor;
out vec4 fragColor;

void main()
{
    fragColor = vec4(outColor,1);
}
```

Demonstration of SimpleApp

– Available on course homepage in Schedule.

– You need OpenGL 3.0 or later

Cool application

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();
}
```

OLD WAY

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.
}
```

OLD WAY

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();
}
```

OLD WAY

Repetition

- What is important:
  - Understand the Application-, Geometry- and Rasterization Stage