TDA362 – Computer Graphics

Teacher: Ulf Assarsson

Chalmers University of Technology
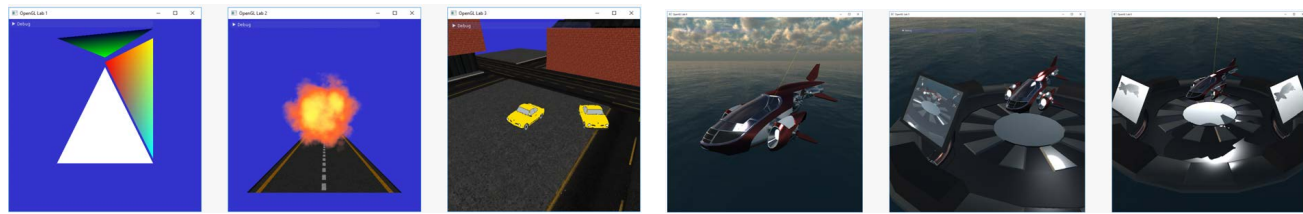
This Course

- Algorithms!

Understanding of Ray Tracing

Real-time Rendering

Course Info

- Study Period 2 (lp2)
- Real Time Rendering, 3rd edition
  - Available on Cremona
- Schedule:
  - Mon 13-15, w2 only
  - Tues 10-12,
  - Fri 9-12, except w7
    - ~14 lectures in total, ~2 / week
  - Labs:
    - Mon: 17-21
    - Tues: 13-21
    - Wed: 13-21
    - Thur: 9-12 + 17-21
- Homepage:
  - Google "TDA362" or
  - "Computer Graphics Chalmers"

| Date	Lecture	Readings/Läsanvisningar	Notes
Week 1			
Tue 31/10	Lecture 1 - Introduction + Pipeline and OpenGL		
Fri 3/11	Lecture 2 - Vectors and Transforms		
Week 2			
Mon 6/11	Lecture 3 - Shading and Antialiasing		
Tue 7/11	Lecture 4 - Texturing		
Fri 10/11	Lecture 5 - OpenGL		
Week 3			
Tue 14/11	Lecture 6 - Intersections		
Fri 17/11	Lecture 7 - Spatial data structures and Collision Detection		
Week 4			
Tue 21/11	Lecture 8 - Ray Tracing 1		
Fri 24/11	Lecture 9 - Ray Tracing 2		
Week 5			
Mon 27/11	**Canceled**		
Tue 28/11	Lecture 10 - Global Illumination		
Fri 1/12	Lecture 11 - Shadows and Reflections		
Week 6			
Tue 5/12	Lecture 12 - Curves		
Fri 8/12	Lecture 13 - Graphics Hardware		
Week 7			
Tue 12/12	Lecture 14 - Repetition		
Fri 15/12	**Canceled**		

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
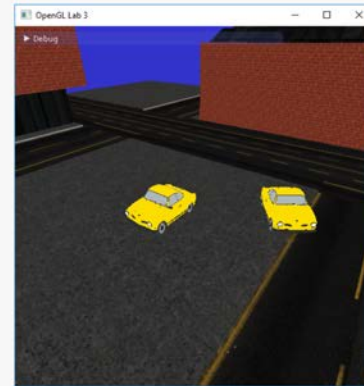 - Real-time rendering
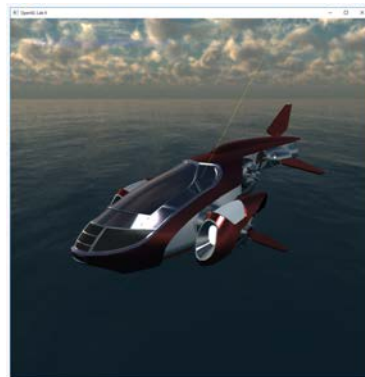
    or
  - Path tracer
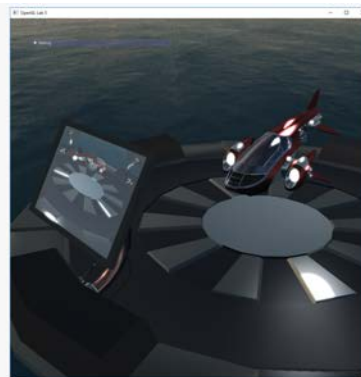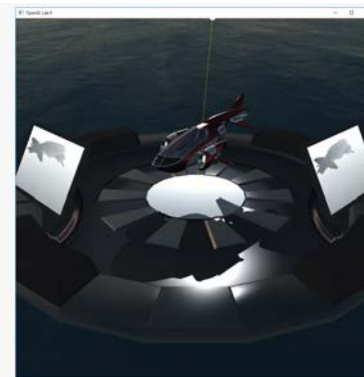
Tutorials 1-6
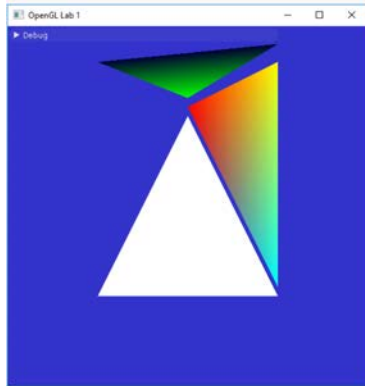
Rendering a triangle

Textures

Animation

Shading

Render to textures

Shadow maps

Project

Choose at least 1 from:

Project

Real-time rendering   Offline rendering
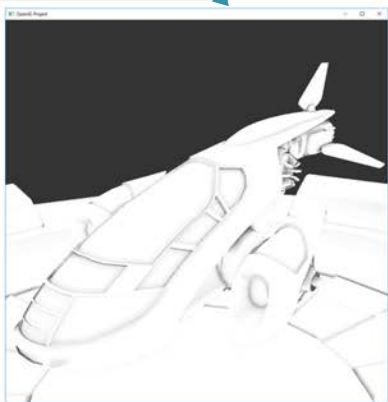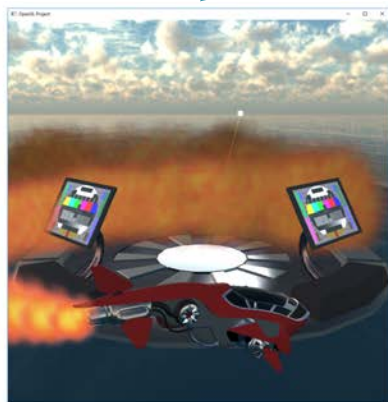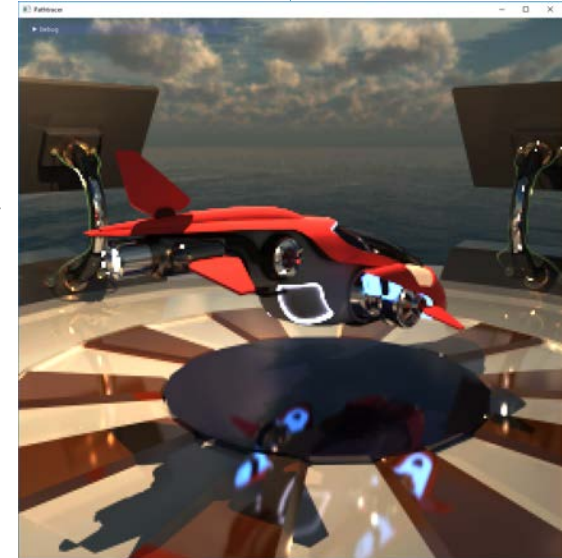
Screen-space ambient occlusion   or   Particle System   or   Height field   or   Path Tracing

Tutorials

- Info: http://www.cse.chalmers.se/edu/course/TDA362/tutorials.html
- Rooms 4225, 4th floor EDIT-building
  - Or your favorite place/home
- Time slots available every day.
  - No booking. First come first served.
- To pass the tutorials:
  - Show your solutions to lab assistants in lab rooms (bring your computer if done at home)
  - Deadlines:
    - Lab 1+2+3: Thursday week 2.
    - Lab 4 + 5: Thursday week 3.
    - Lab 6: Thursday week 4
    - Lab 7 / Project: Thursday week 7.
- Do the tutorials in groups (Labgrupper) of two, 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.

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.
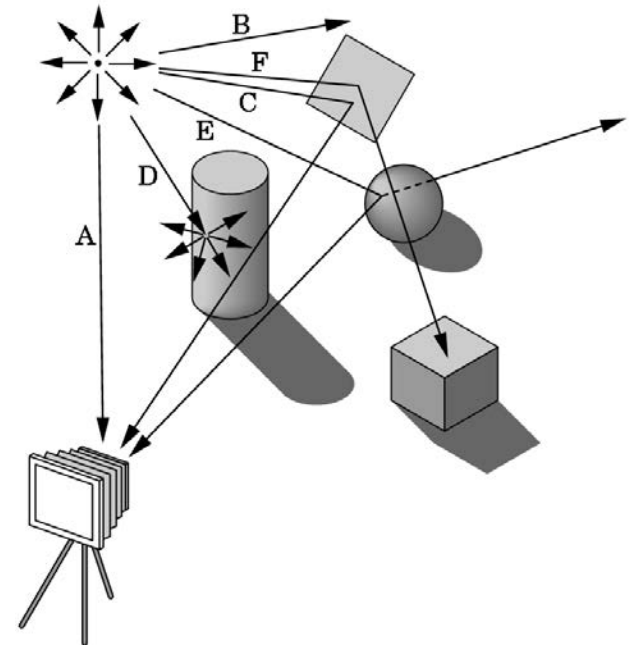
Other Physical Approaches

- **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 rather slow
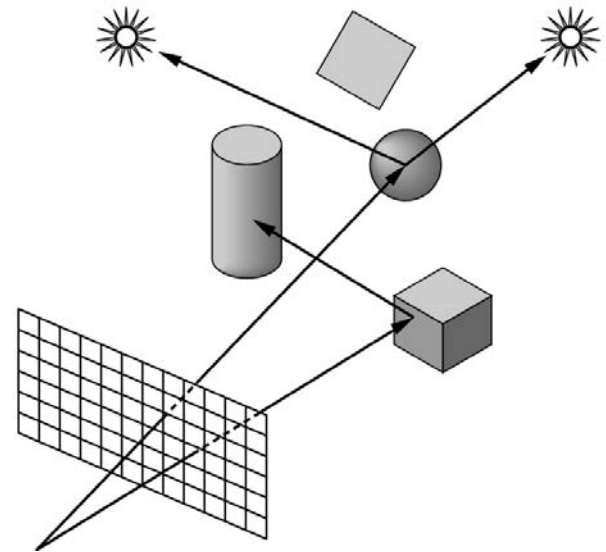
Real-Time Rendering

Overview of the
Graphics Rendering Pipeline
and OpenGL

Real-Time Rendering

The screen consists of pixels
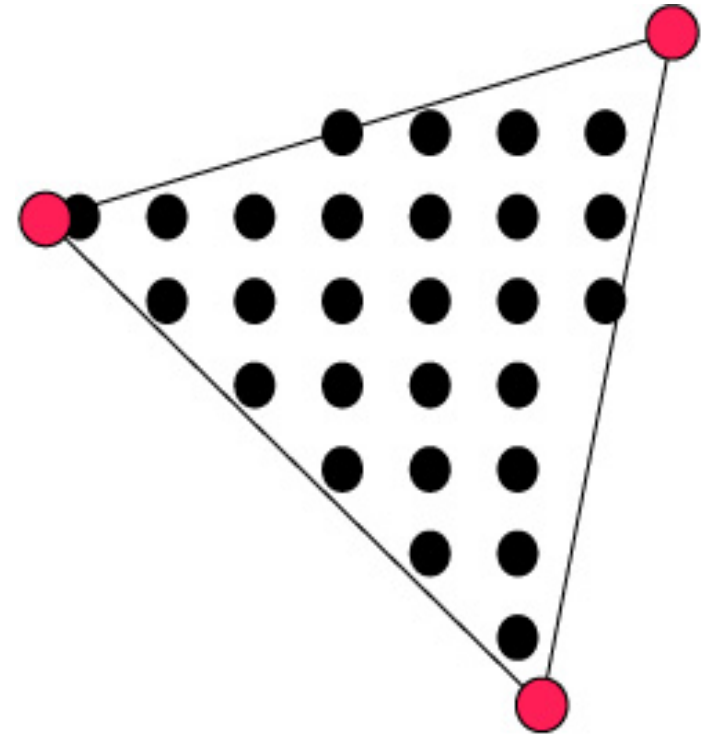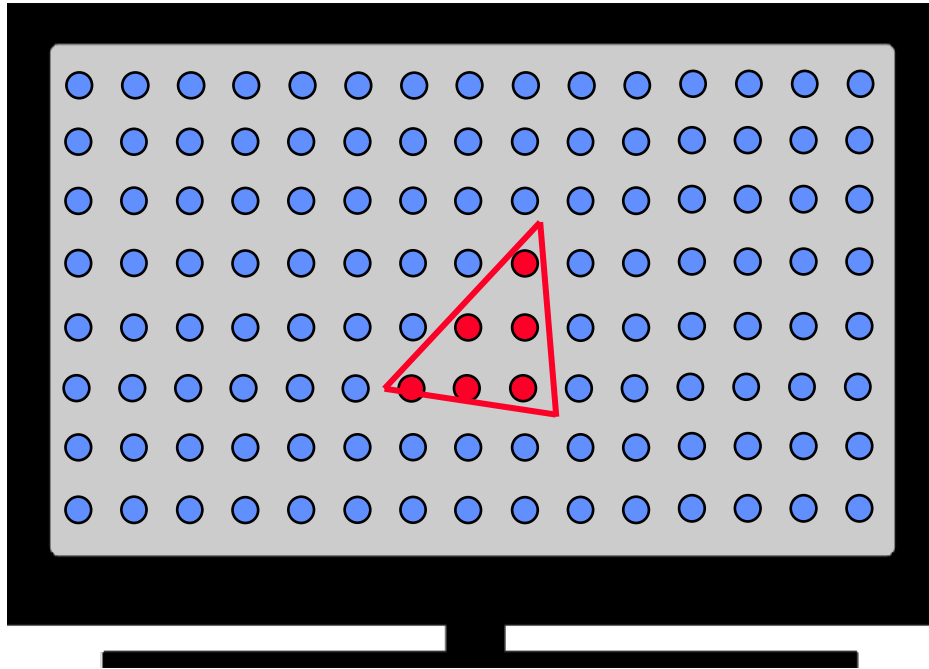
Graphics Cards
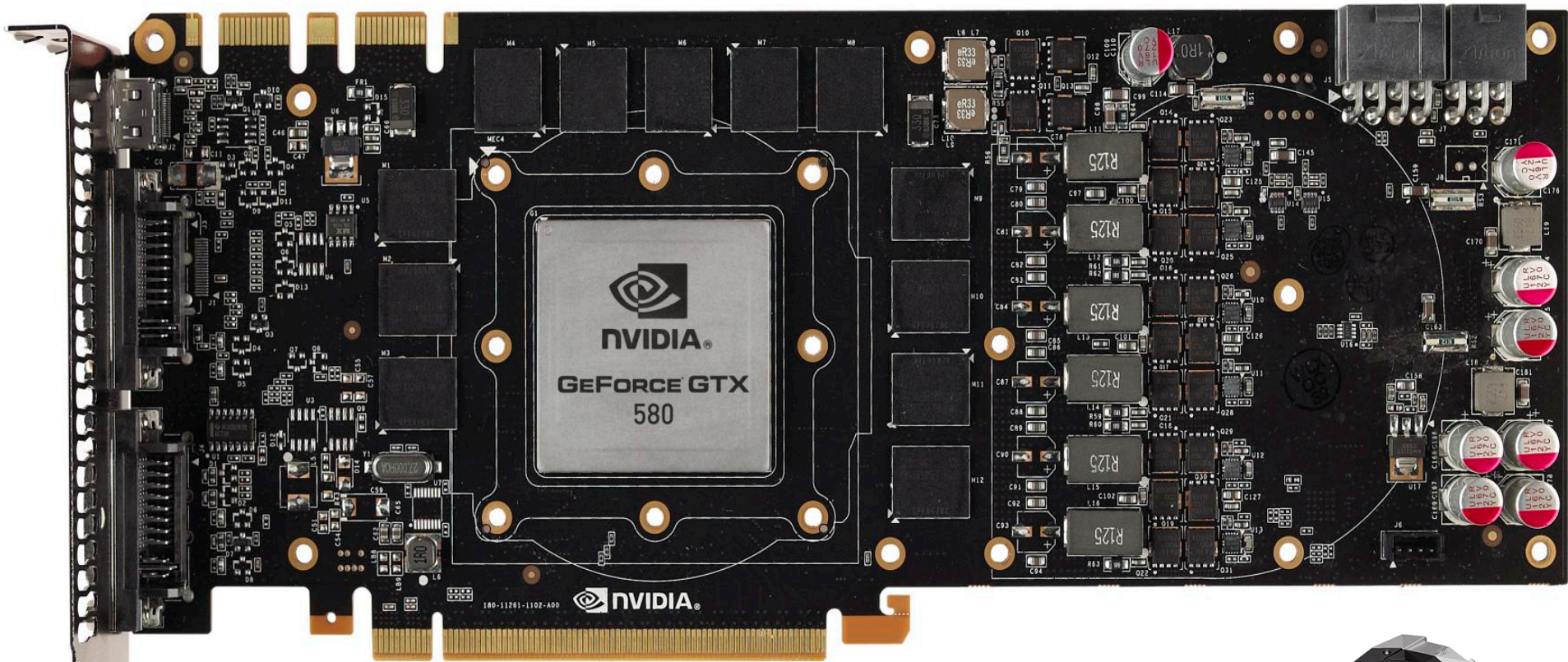
3D-Rendering

- 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

Why only triangles?

Y

X

Z
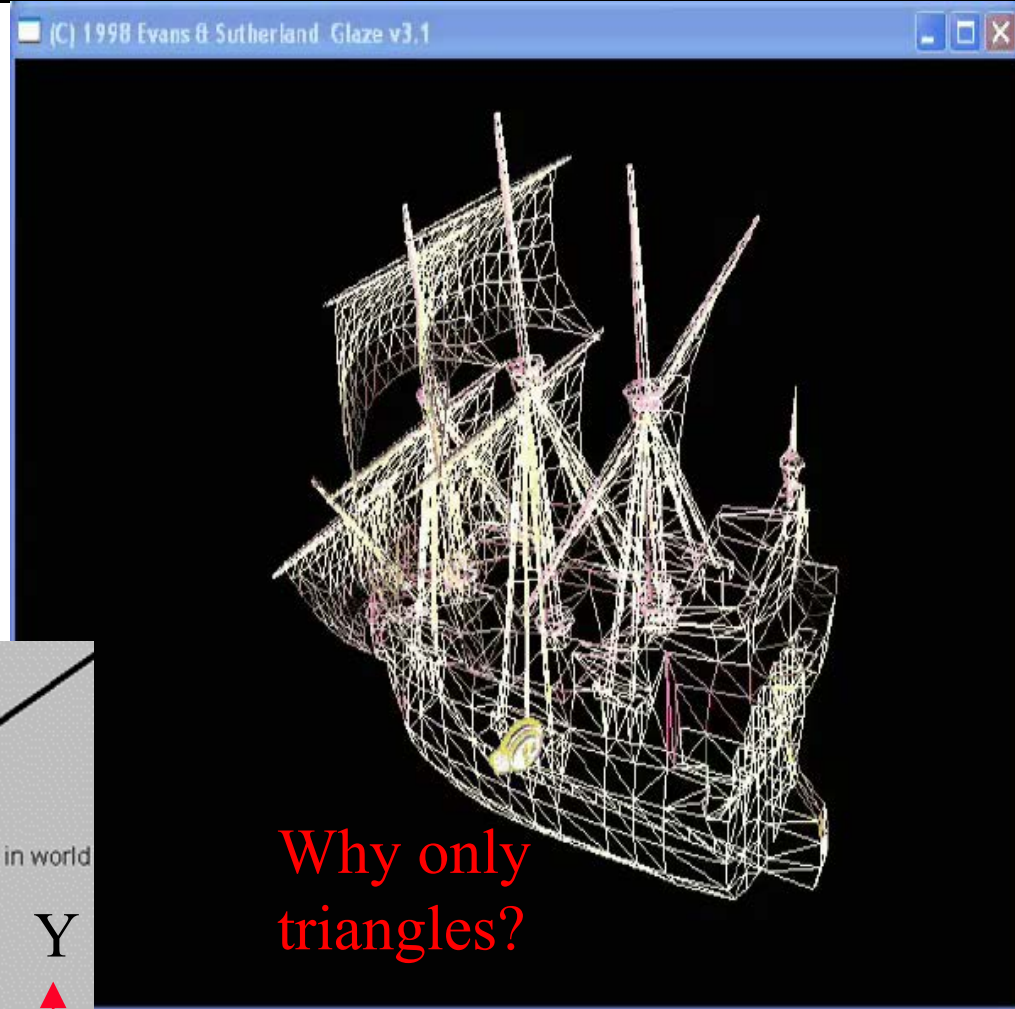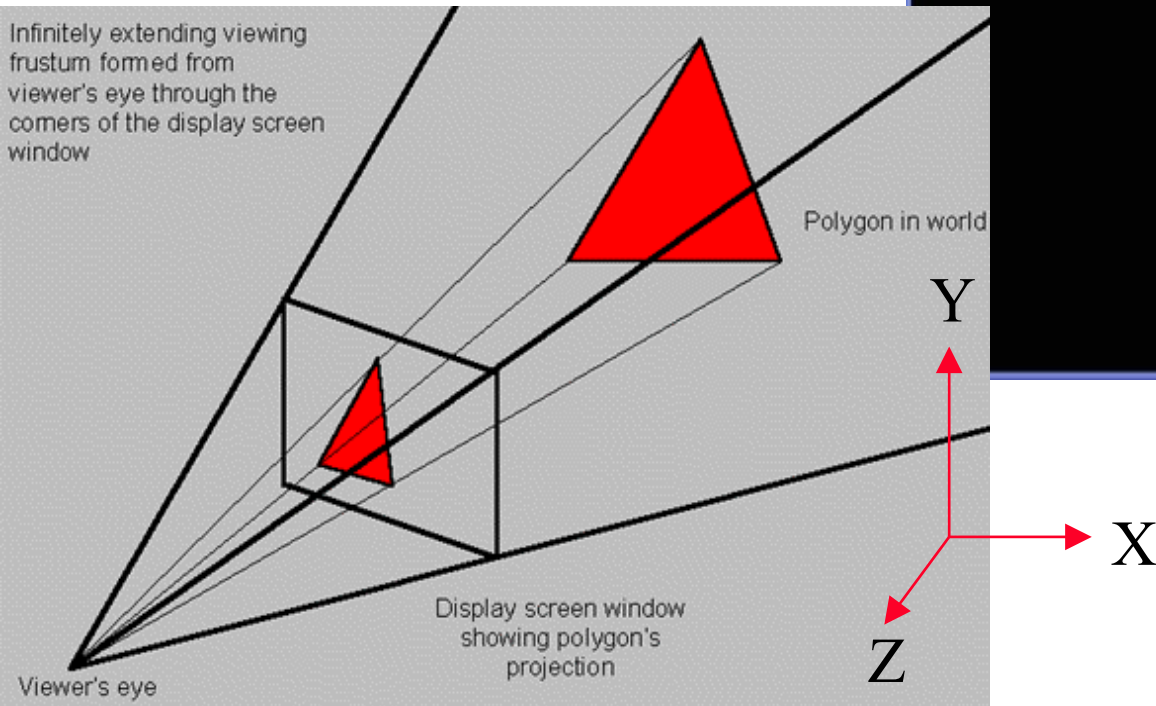
4D Matrix Multiplication

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

Real-Time Rendering
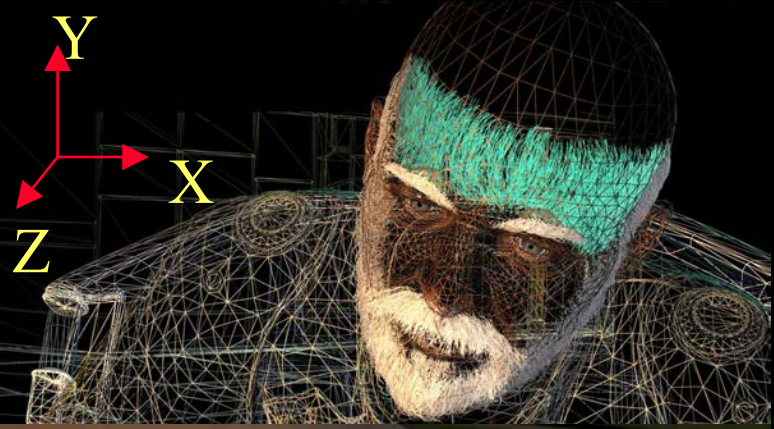
Textures

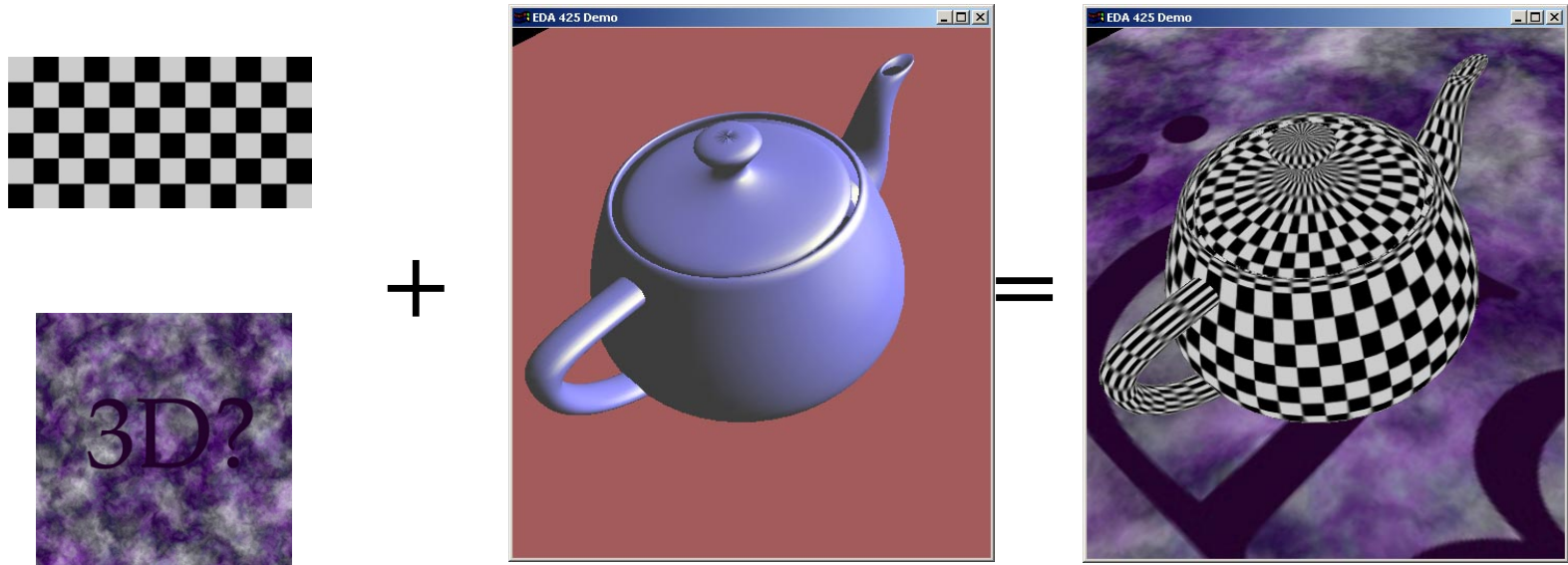- One application of texturing is to "glue" images onto geometrical object
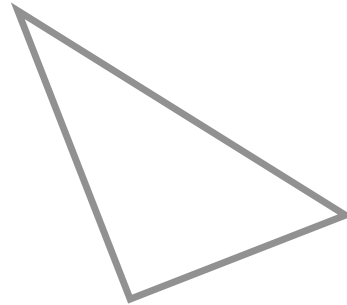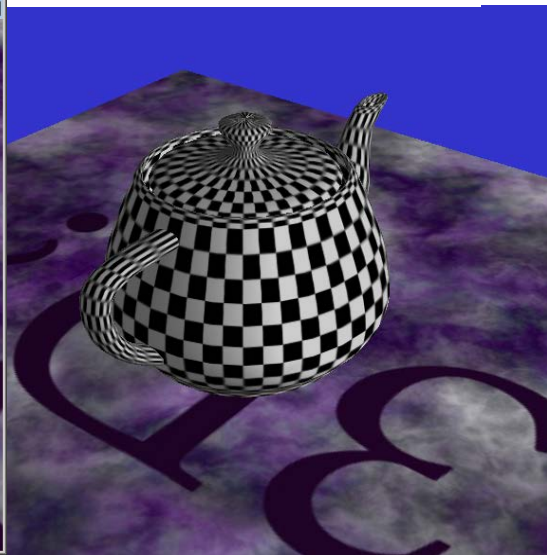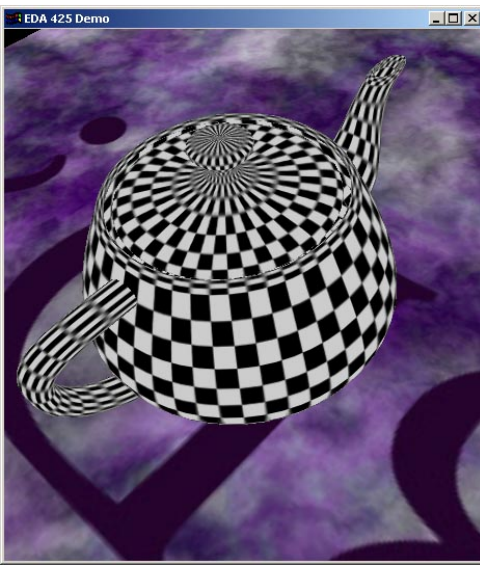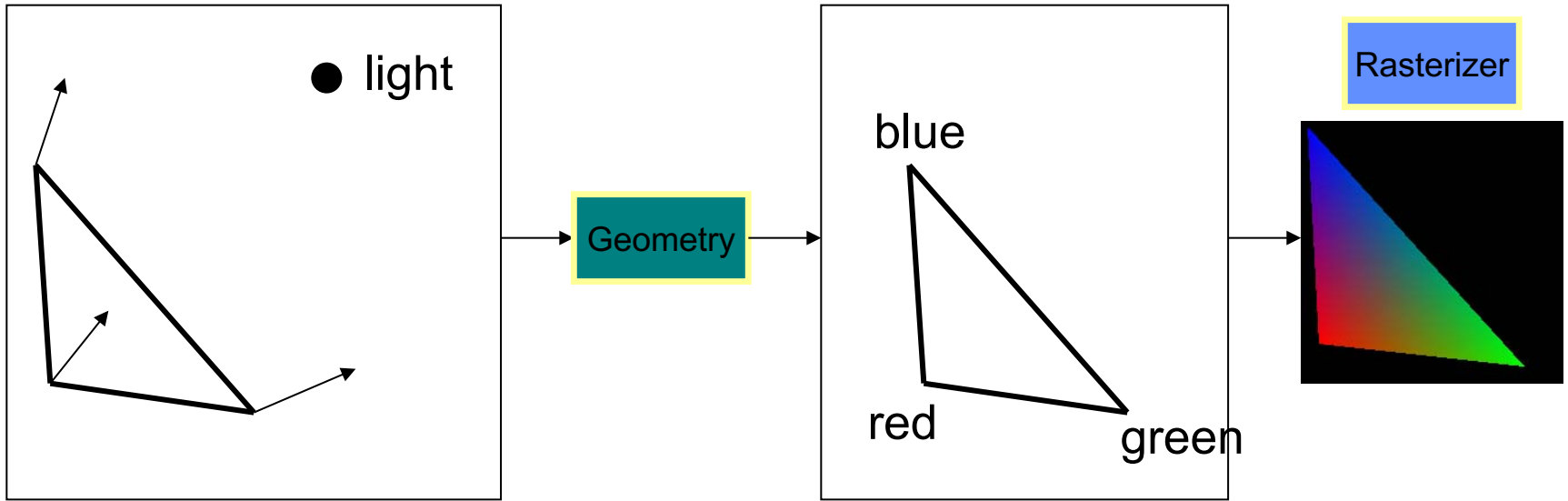
Texturing: Glue images onto geometrical objects

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

Lighting computation per triangle vertex

The Graphics Rendering Pipeline

The Application stage, geometry stage, and rasterizer stage

# 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
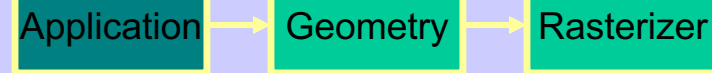 - A vertex is 3D position, and may include a normal, color, texture coordinate, ….
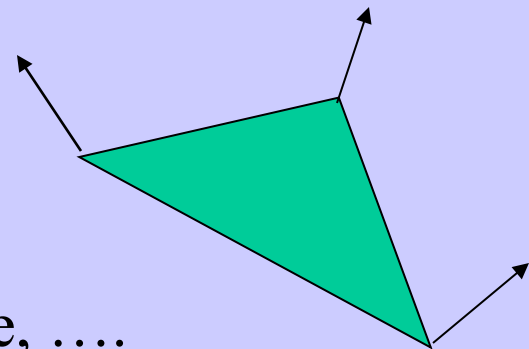
Lecture 1: Real-time Rendering
The Graphics Rendering Pipeline

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

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

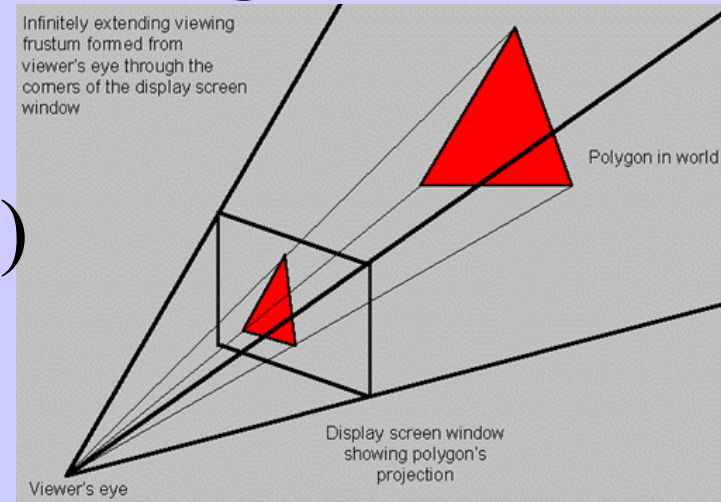| Application | → | Geometry | → | Rasterizer | → | Image |

input → 3D scene

output
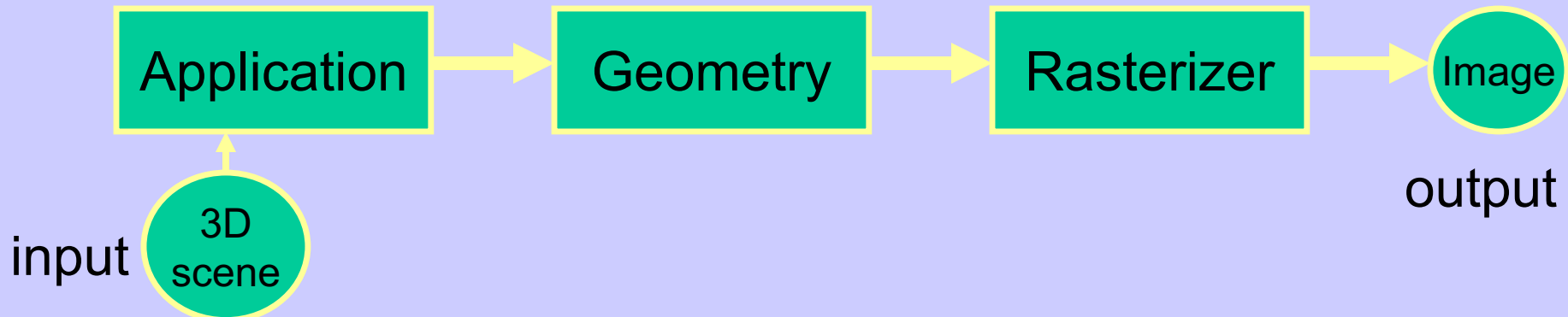
The APPLICATION stage

- 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

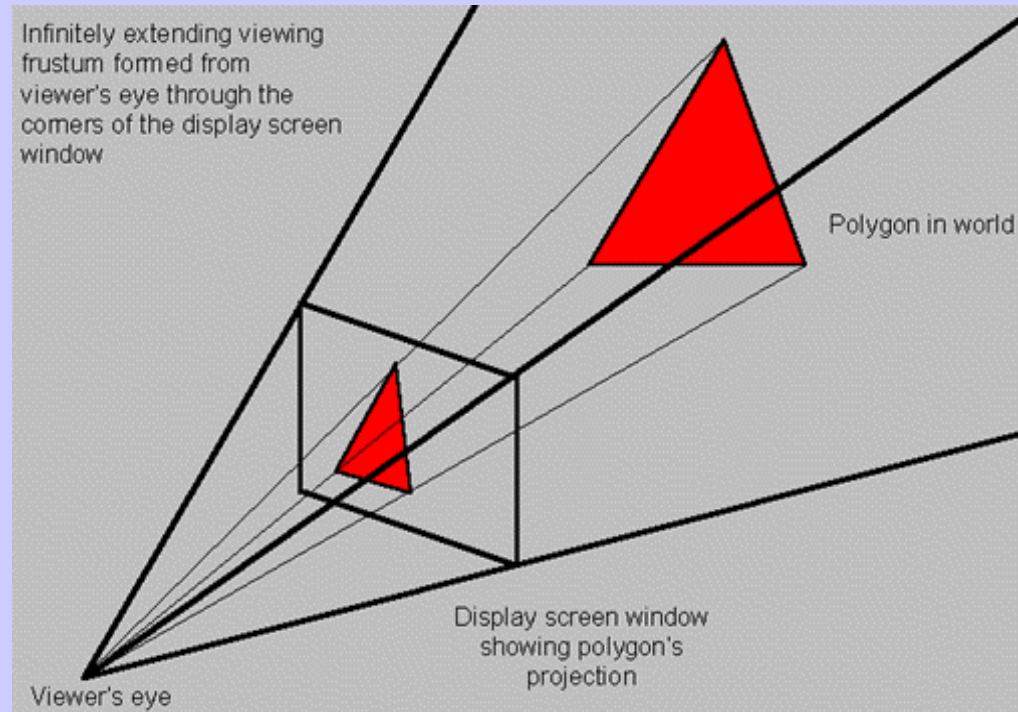Model & View Transform → Vertex Shading → Projection → Clipping → Screen Mapping

- (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
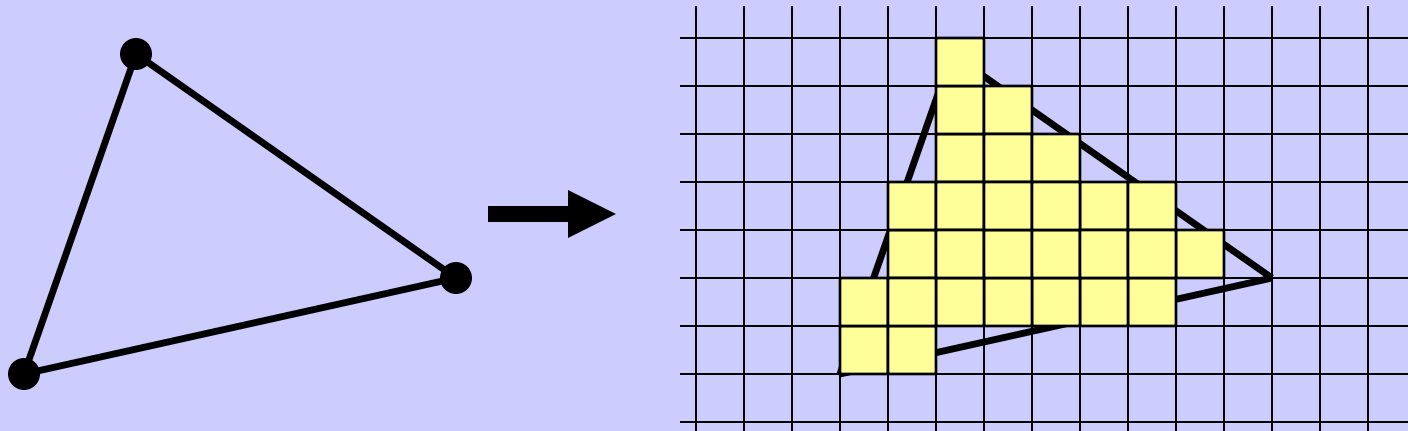
- Computes color per pixel, using fragment shader (=pixel shader)

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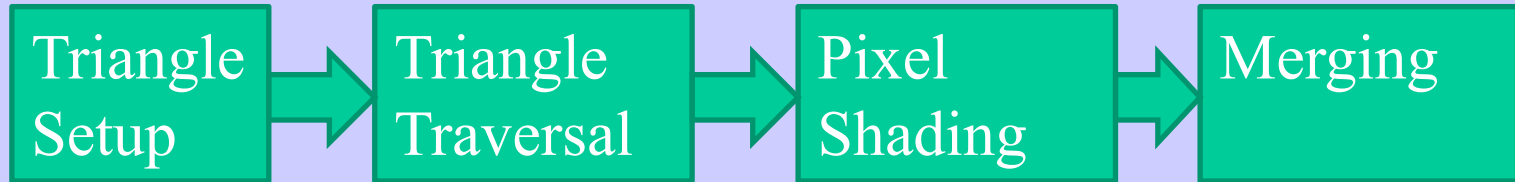
Triangle Setup:
- collect three vertices + vertex shader output (incl. normals) and make one triangle.

Triangle Traversal
- Scan conversion

Pixel Shading
- Compute pixel color

Merging:
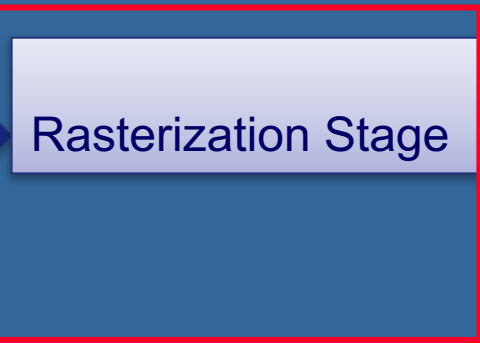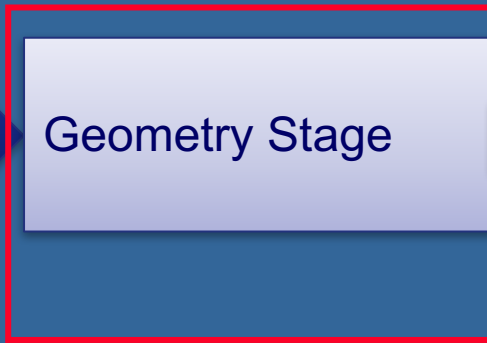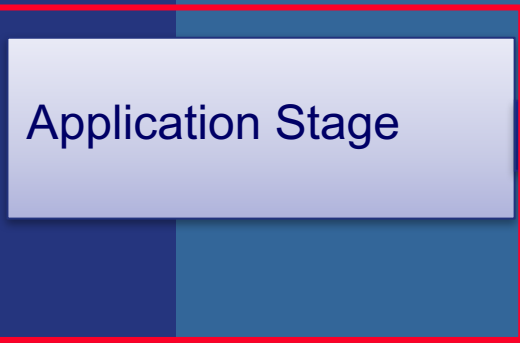- output color to screen

The three stages' correlation to hardware

The Application stage, geometry stage, and rasterizer stage

Rendering Pipeline and Hardware

CPU

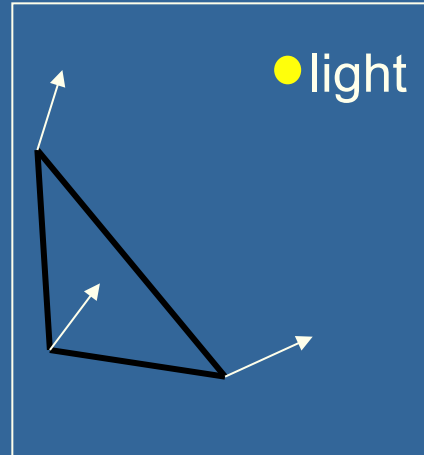GPU

Application Stage →

Geometry Stage →

Rasterization Stage

Rendering Pipeline and Hardware

CPU

GPU

| Appli-cation Stage | Geometry Stage | Rasterization Stage |

HARDWARE

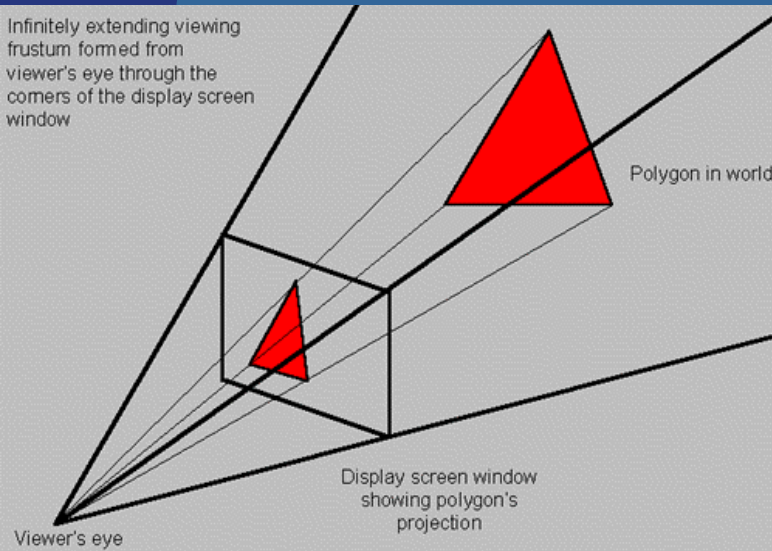| Vertex shader | Geometry shader | Clipping | Screen mapping | Triangle Setup | Triangle Traversal | Pixel shader | Merger |

Display

Hardware design
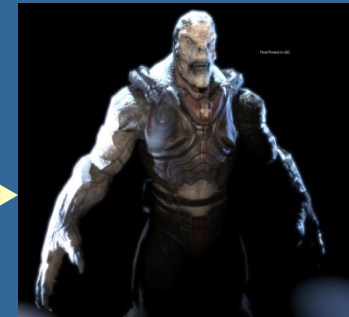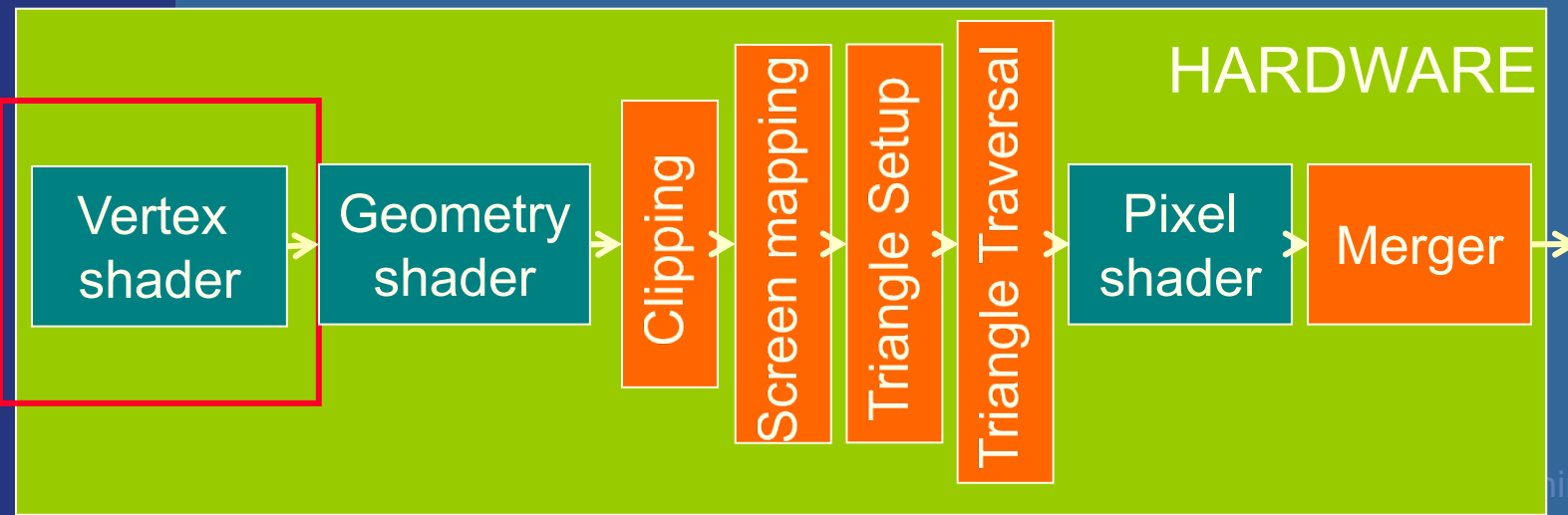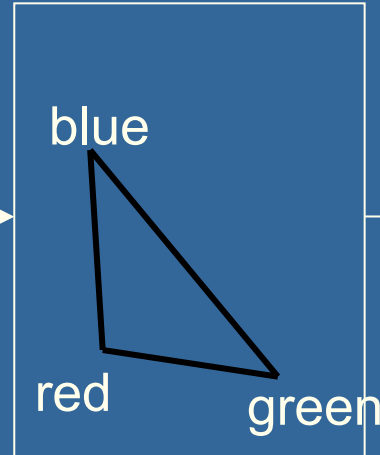
Vertex shader:

- 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

blue

red

green

Geometry

HARDWARE

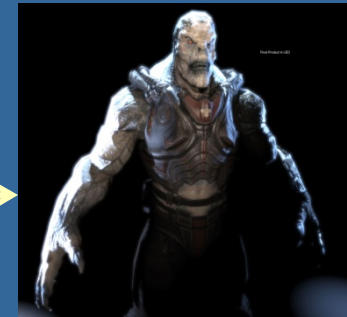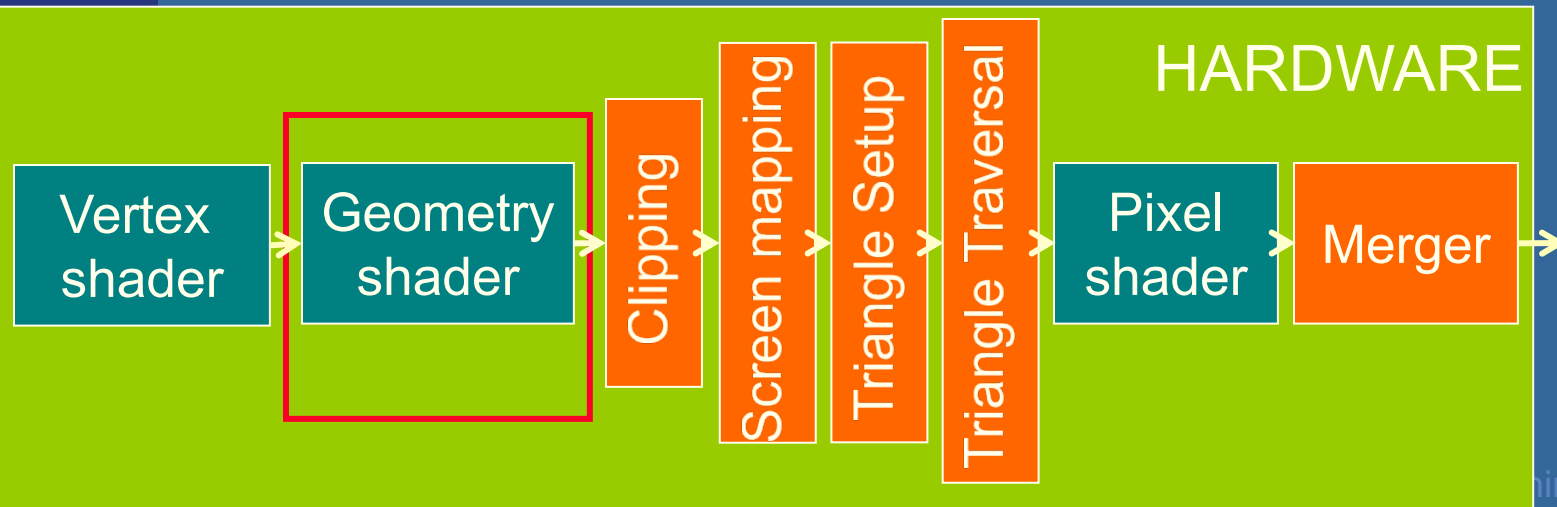| Vertex shader | Geometry shader | Clipping | Screen mapping | Triangle Setup | Triangle Traversal | Pixel shader | Merger |

Display

Akenine-Möller © 2003

Hardware design

Geometry shader:

- One input primitive

- Many output primitives

or

HARDWARE

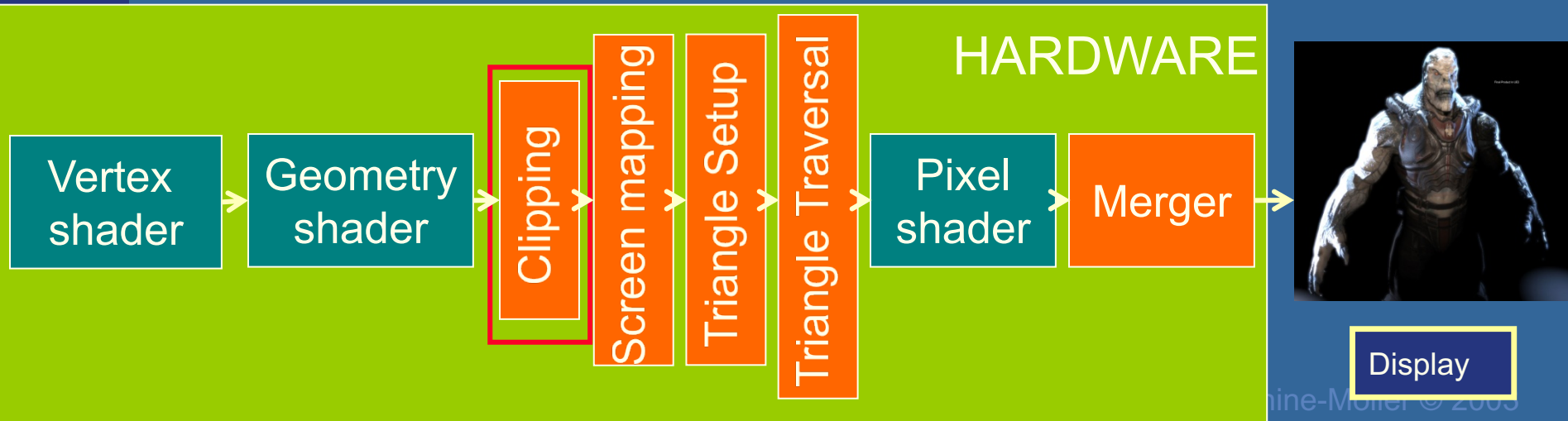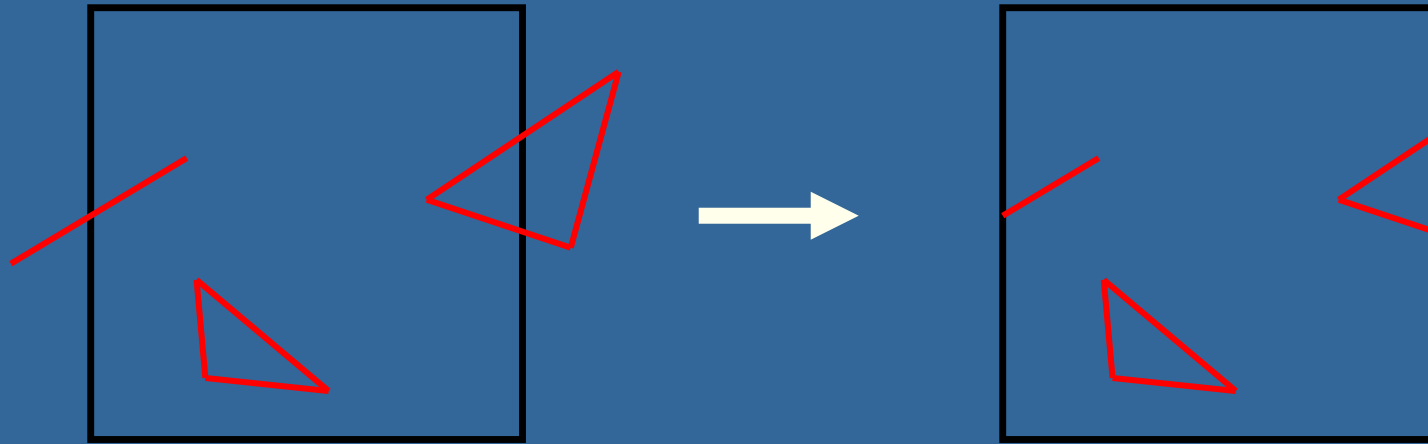| Vertex shader | Geometry shader | Clipping | Screen mapping | Triangle Setup | Triangle Traversal | Pixel shader | Merger |

Display

Akenine-Möller © 2003

Hardware design

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

HARDWARE

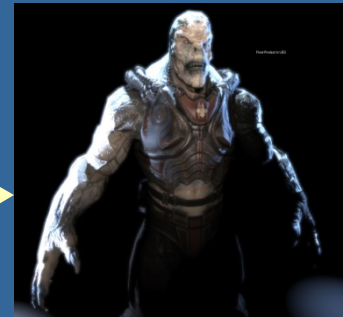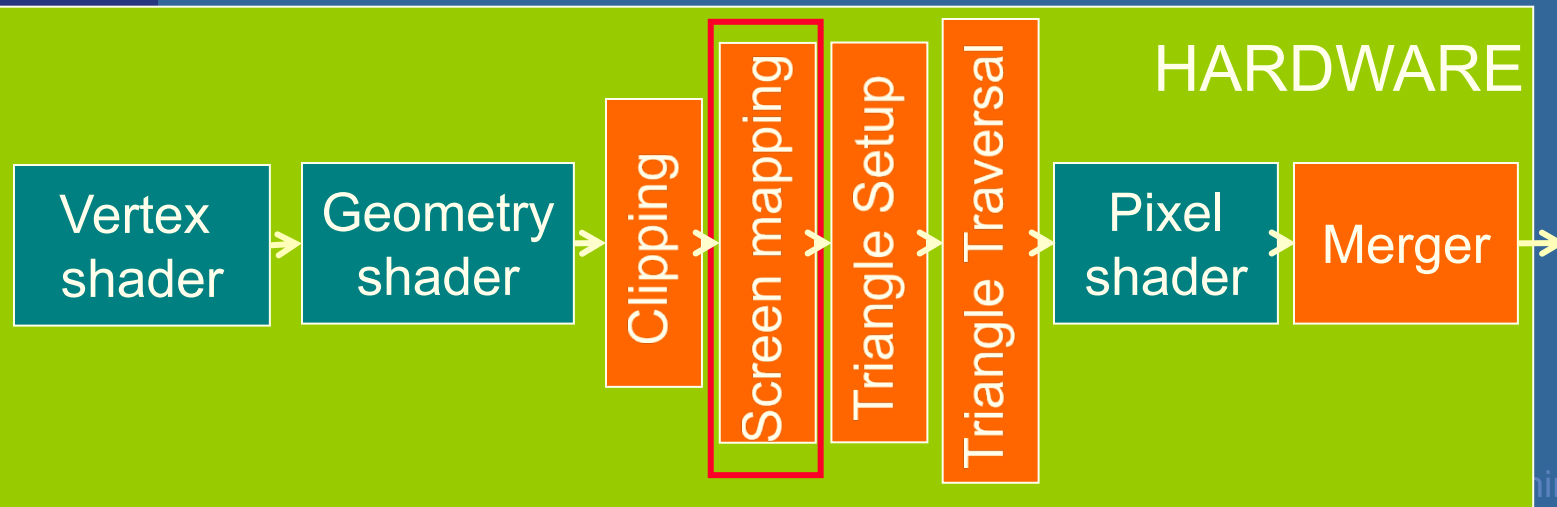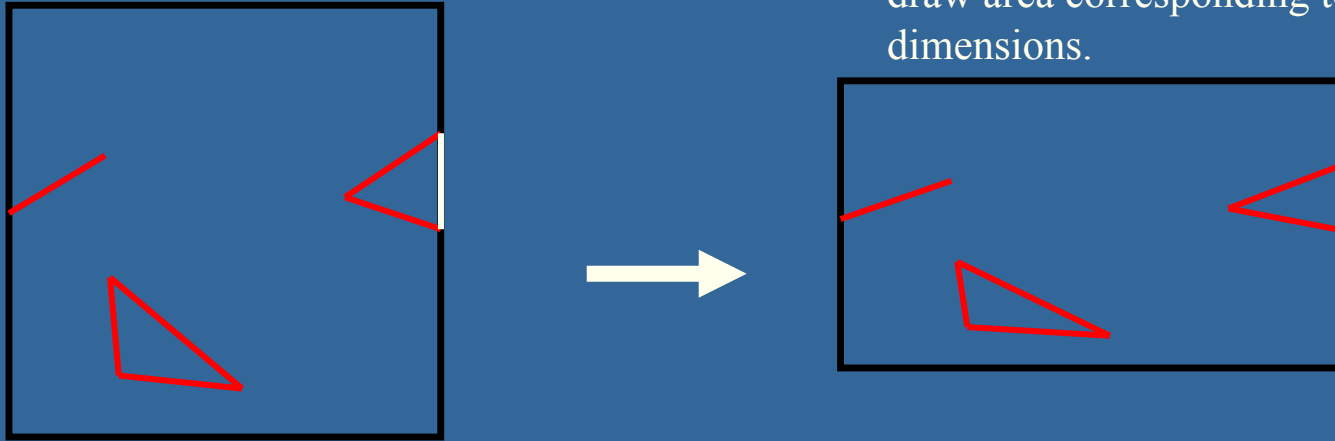| Vertex shader | Geometry shader | Clipping | Screen mapping | Triangle Setup | Triangle Traversal | Pixel shader | Merger |

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

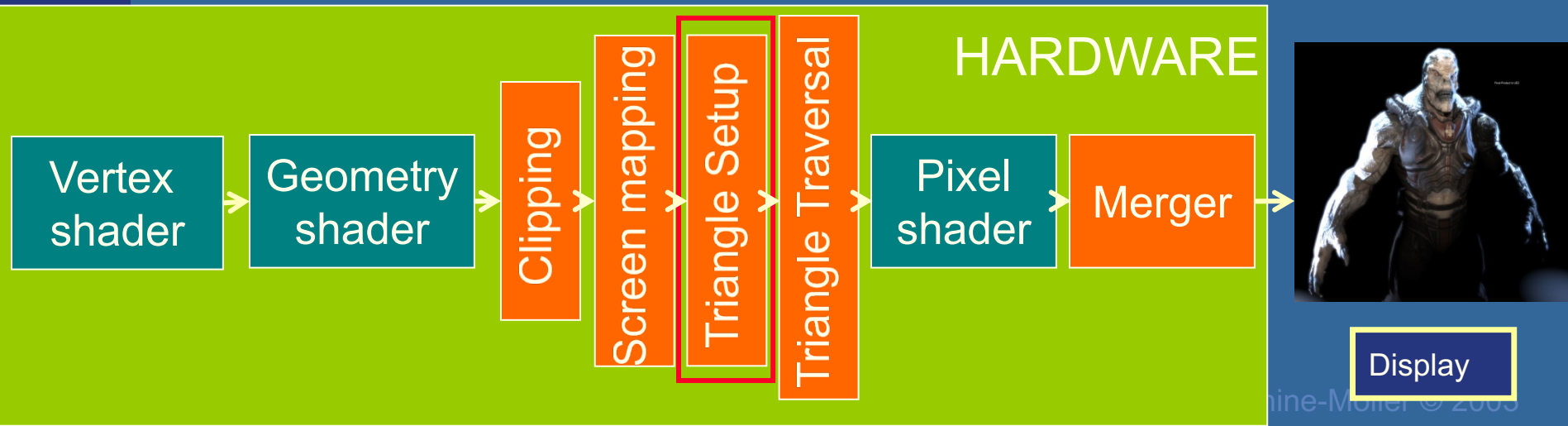| Vertex shader | Geometry shader | Clipping | Screen mapping | Triangle Setup | Triangle Traversal | Pixel shader | Merger |

Display

Akenine-Möller © 2003

Hardware design

Collects three vertices into one triangle

HARDWARE

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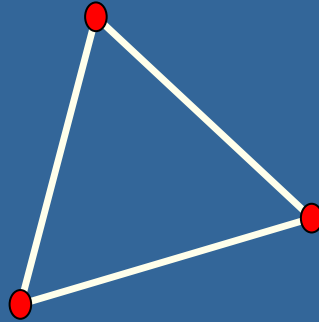
Display

Hardware design

Creates the fragments/pixels for the triangle
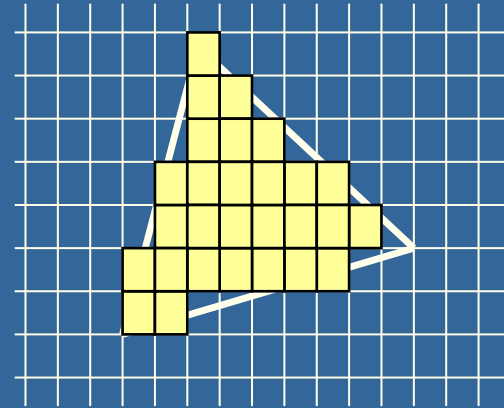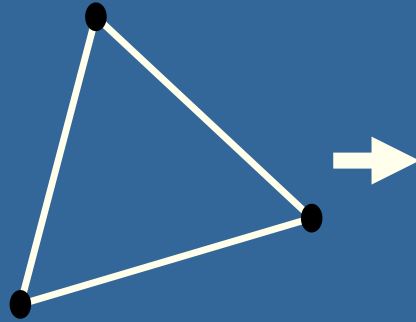
HARDWARE

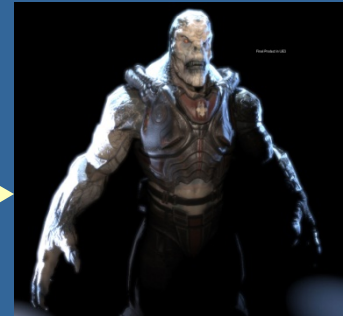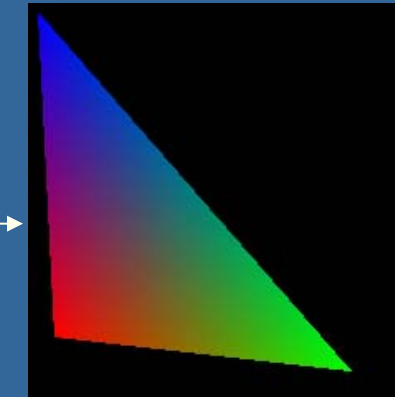| Vertex shader | Geometry shader | Clipping | Screen mapping | Triangle Setup | Triangle Traversal | Pixel shader | Merger |

Display

Hardware design

blue

red

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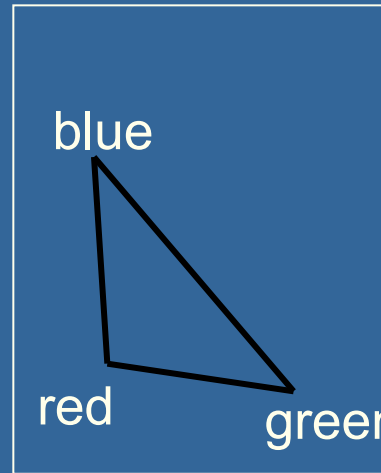
Rasterizer
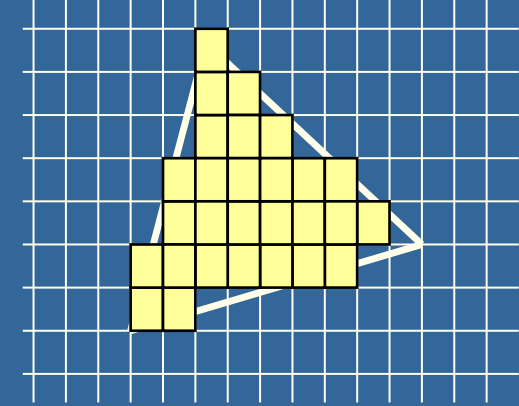
Pixel Shader:
Compute color using:
•Textures
•Interpolated data (e.g. Colors + normals) from vertex shader

HARDWARE

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

Display

Ahine-Moller © 2003
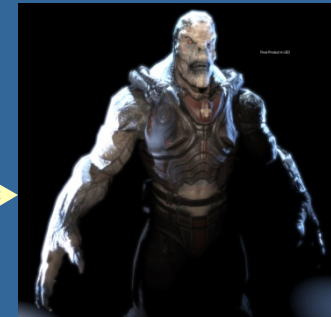
Hardware design

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

Hunter ate Ranger's rocket
Wrote screenshots/shot0147.tga

Frame buffer:

- Color buffers
- Depth buffer
- Stencil buffer

HARDWARE

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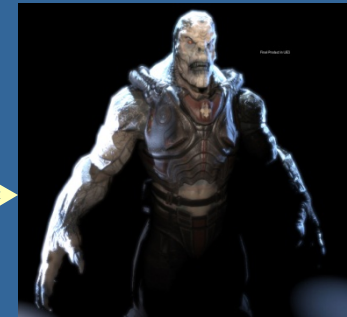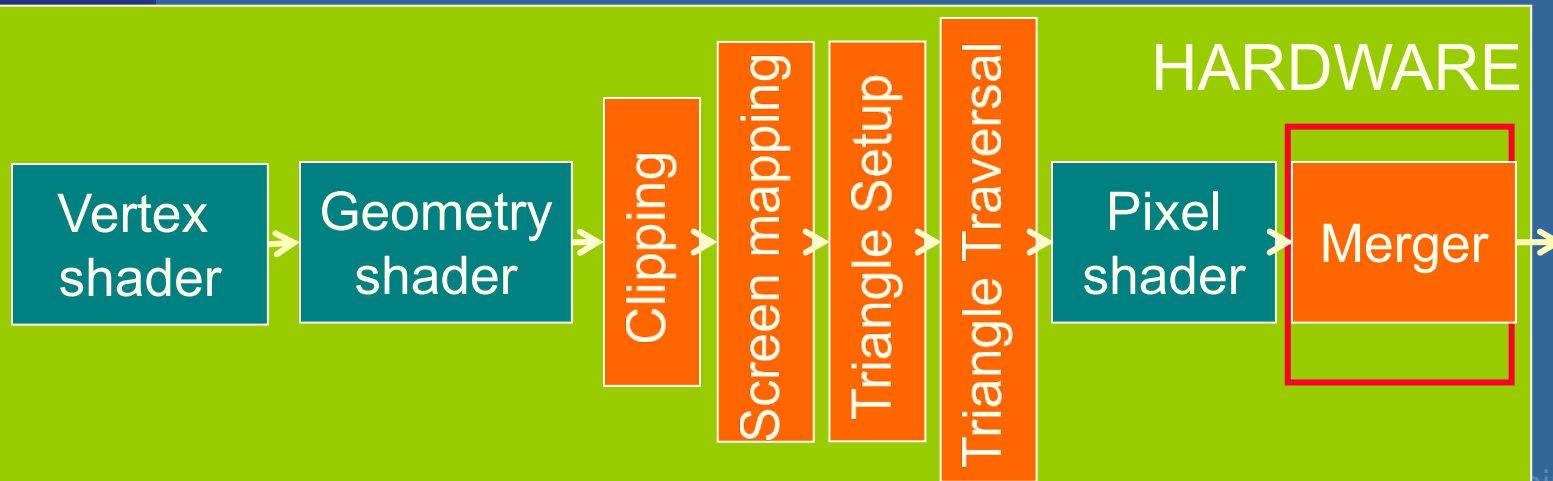
Display

What is vertex and fragment (pixel) shaders?
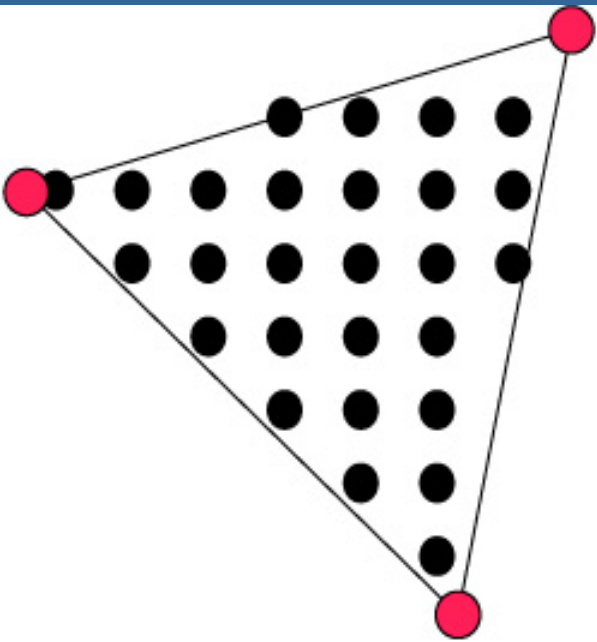
- Vertex shader: reads from textures Writes output data per vertex, which are interpolated and input to each fragment shader invocation.

- Fragment shader: reads from textures, writes to pixel color

- Memory: Texture memory (read + write) typically 1 GB – 8 GB

- Program size: the smaller the faster

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

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

Shaders
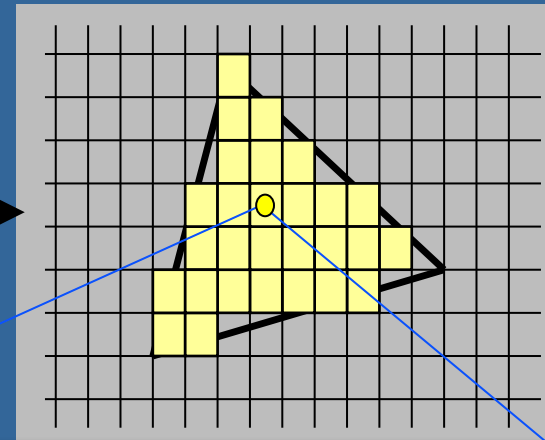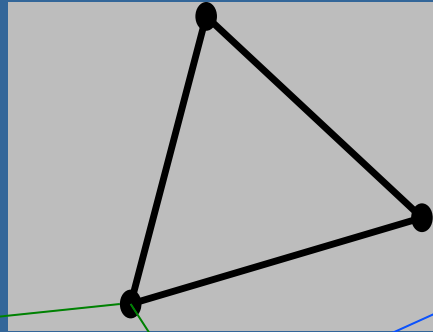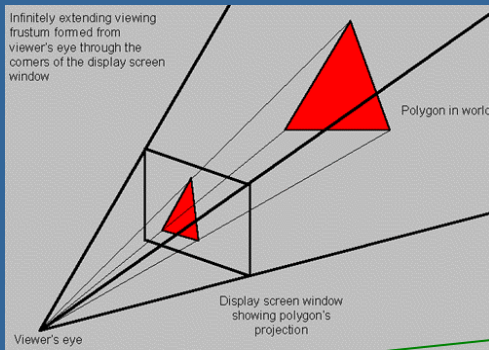

```glsl
// Vertex Shader
#version 420

layout(location = 0) in vec3 vertex;
layout(location = 1) in vec3 color;
out vec3 outColor;
uniform mat4 modelViewProjectionMatrix;

void main()
{
    gl_Position = modelViewProjectionMatrix *
            vec4(vertex,1);
    outColor = color;
}
```
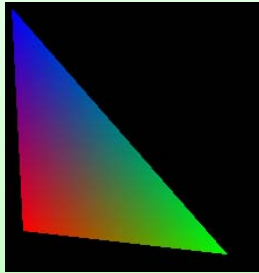
```glsl
// Fragment Shader:
#version 420

precision highp float;

in  vec3 outColor;

layout(location = 0) out vec4 fragColor;
// Here, location=0 means that we draw to
//     frameBuffer[0], i.e., the screen.

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

Shaders

```
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;
}
```
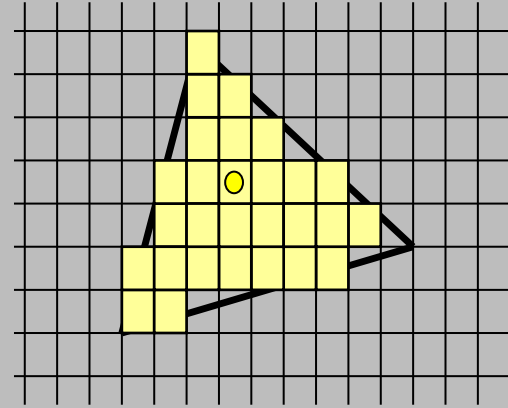
OpenGL

CPU-side

Language: C++

API: OpenGL (Direct3D)

Window system: SDL (Cocoa, Win32,…)

```cpp
C++:
float positions[] = {
//   X        Y       Z
    0.0f,   0.5f, 1.0f, // v0
   -0.5f,  -0.5f, 1.0f, // v1
    0.5f,  -0.5f, 1.0f  // v2
    ...                 // ... vn
};
// and any other per-vertex data, e.g.:
float colors[] = {
//   R        G       B
    1.0f,   0.0f, 0.0f, // c0
   -0.0f,   1.0f, 0.0f, // c1
    0.0f,   0.0f, 1.0f  // c2
    ...                 // ... cn
};
float normals[] = {...};
float textureCoords[] = {...};
```
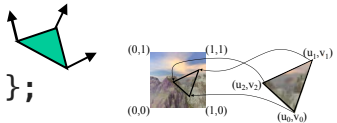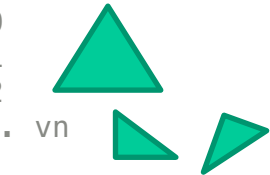
```cpp
OpenGL:
        Vertex-buffer objects
uint32 positionsBuffer; // type: GL_ARRAY_BUFFER
uint32 colorsBuffer;    // type: GL_ARRAY_BUFFER

        Vertex-Array object
uint32 vertexArrayObject;

        Shaders
uint32 vertexShader;
uint32 fragmentShader;
uint32 shaderProgram;
```

GPU-side

Language: GLSL

used for vertex- ,geometry-, and fragment shaders

```glsl
Vertex Shader:
#version 420

layout(location = 0) in  vec3 position;
layout(location = 1) in  vec3 color;

out vec3 outColor;

void main()
{
    gl_Position = vec4(position, 1.0);
    outColor = color;
}
```
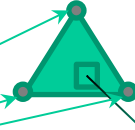
```glsl
Fragment Shader:
#version 420

precision highp float; // required by GLSL spec Sect 4.5.3
                       // (though nvidia does not, amd does)

layout(location = 0) out vec4 fragmentColor;
in vec3 outColor;          Per-pixel-interpolated value

void main()
{
    // fragmentColor = vec4(1,1,1,1);
    fragmentColor.rgb = outColor;
    fragmentColor.a = 1.0;
}
```

CPU-side

Language:　　　 C++
API:　　　　　 OpenGL (Direct3D)
Window system: SDL (Cocoa, Win32,…)

```cpp
C++:
float positions[] = {
//   X        Y      Z
   0.0f,   0.5f, 1.0f, // v0
  -0.5f,  -0.5f, 1.0f, // v1
   0.5f,  -0.5f, 1.0f  // v2
   ...                 // ... vn
};
// and any other per-vertex data, e.g.:
float colors[] = {
//   R       G      B
   1.0f    0.0f   0.0f   // c0
```

```
How to connect the
vertexArrayObject as vertex
shader input (position, color)
```

```cpp
};
flo
flo
```

```cpp
OpenGL:
        Vertex-buffer objects
uint32 positionsBuffer; // type: GL_ARRAY_BUFFER
uint32 colorsBuffer;    // type: GL_ARRAY_BUFFER

        Vertex-Array object
uint32 vertexArrayObject;

        Shaders
uint32 vertexShader;
uint32 fragmentShader;
uint32 shaderProgram;
```

GPU-side

Language: GLSL
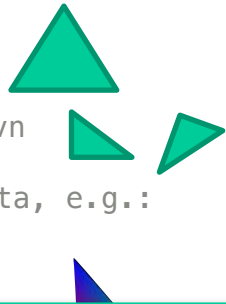　　　　 used for vertex- ,geometry-, and
　　　　 fragment shaders

```glsl
Vertex Shader:
#version 420

layout(location = 0) in  vec3 position;
layout(location = 1) in  vec3 color;

out vec3 outColor;

void main()
{
    gl_Position = vec4(position, 1.0);
    outColor = color;
}
```

```cpp
glGenVertexArrays(1, &vertexArrayObject);
// Following calls affect this vertex array object.
glBindVertexArray(vertexArrayObject);

// Makes positionBuffer the current array buffer for subse
quent calls.
glBindBuffer( GL_ARRAY_BUFFER, positionBuffer );
// Attach positionBuffer to vertexArrayObject,
// in attribute location 0
glVertexAttribPointer(0, 3, GL_FLOAT, …);

// Makes colorBuffer the current array buffer for subseque
nt calls.
glBindBuffer( GL_ARRAY_BUFFER, colorBuffer );
// Attaches colorBuffer to vertexArrayObject,
// in attribute location 1
glVertexAttribPointer(1, 3, GL_FLOAT, … );

glEnableVertexAttribArray(0); // Enable attribute array 0
glEnableVertexAttribArray(1); // Enable attribute array 1
```
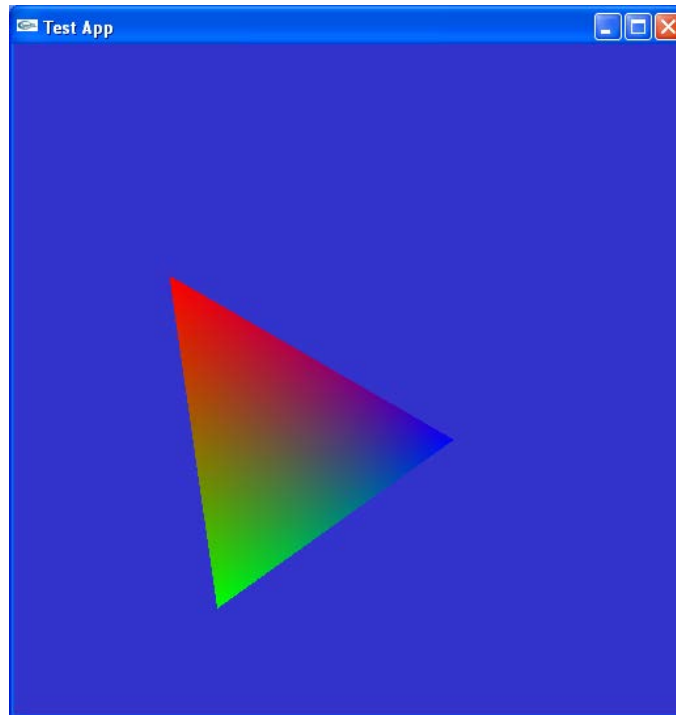
A Simple Program
Computer Graphics version of "Hello World"

Generate a triangle on a solid background

Simple Application...

```
int main(int argc, char *argv[])
{
    // open window of size 512x512 with double buffering, RGB colors, and Z-buffering
    g_window = labhelper::init_window_SDL("OpenGL Lab 1", 512, 512);
    initGL();
    while (true) {

        display(); // render our geometry

        SDL_GL_SwapWindow(g_window); // swap front/back buffer. Ie., displays the frame.


        SDL_Event event;
        while (SDL_PollEvent(&event)) {
            if (event.type == SDL_QUIT || (event.type == SDL_KEYUP &&
                event.key.keysym.sym == SDLK_ESCAPE)) {
                    labhelper::shutDown(g_window);
                    return 0;
            }
        }
    }
    return 0;
}
```

```c
void display(void)
{
    // The viewport determines how many pixels we are rasterizing to
    int w, h;
    SDL_GetWindowSize(g_window, &w, &h);
    glViewport(0, 0, w, h);   // Set viewport

    // Clear background
    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

    glDisable(GL_CULL_FACE); // Both front and back face of triangles should be visible

    // DRAW OUR TRIANGLE(S)
    glUseProgram( shaderProgram );  // Shader Program. Sets what vertex/fragment shaders to use.
    // Bind the vertex array object that contains all the vertex data.
    glBindVertexArray(vertexArrayObject);
    // Submit triangles from currently bound vertex array object.
    glDrawArrays( GL_TRIANGLES, 0, 3 );       // Render 1 triangle

    glUseProgram( 0 );  // "unsets" the current shader program. Not really necessary.
}
```

Lab 1 will teach you this, i.e., setting up a shader program and vertex arrays.

Example of a simple GfxObject class

```
class GfxObject {
public:
        load("filename"); // Creates m_shaderProgram + m_vertexArrayObject
        render()
        {
                /* You may want to initiate more OpenGL states, e.g., for
                    textures (more on that in further lectures) */
                glUseProgram(m_shaderProgram);

                glBindVertexArray(m_vertexArrayObject);

                glDrawArrays( GL_TRIANGLES, 0, numVertices);
        };
private:
        uint      numVertices;
        Gluint    m_shaderProgram;
        GLuint    m_vertexArrayObject;
};


Example:
GfxObject myCoolObject;
myCoolObject.load("filename");


In display():
        myCoolObject.render();
```

The Geometry stage and Rasterizer stage in more detail
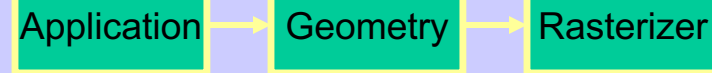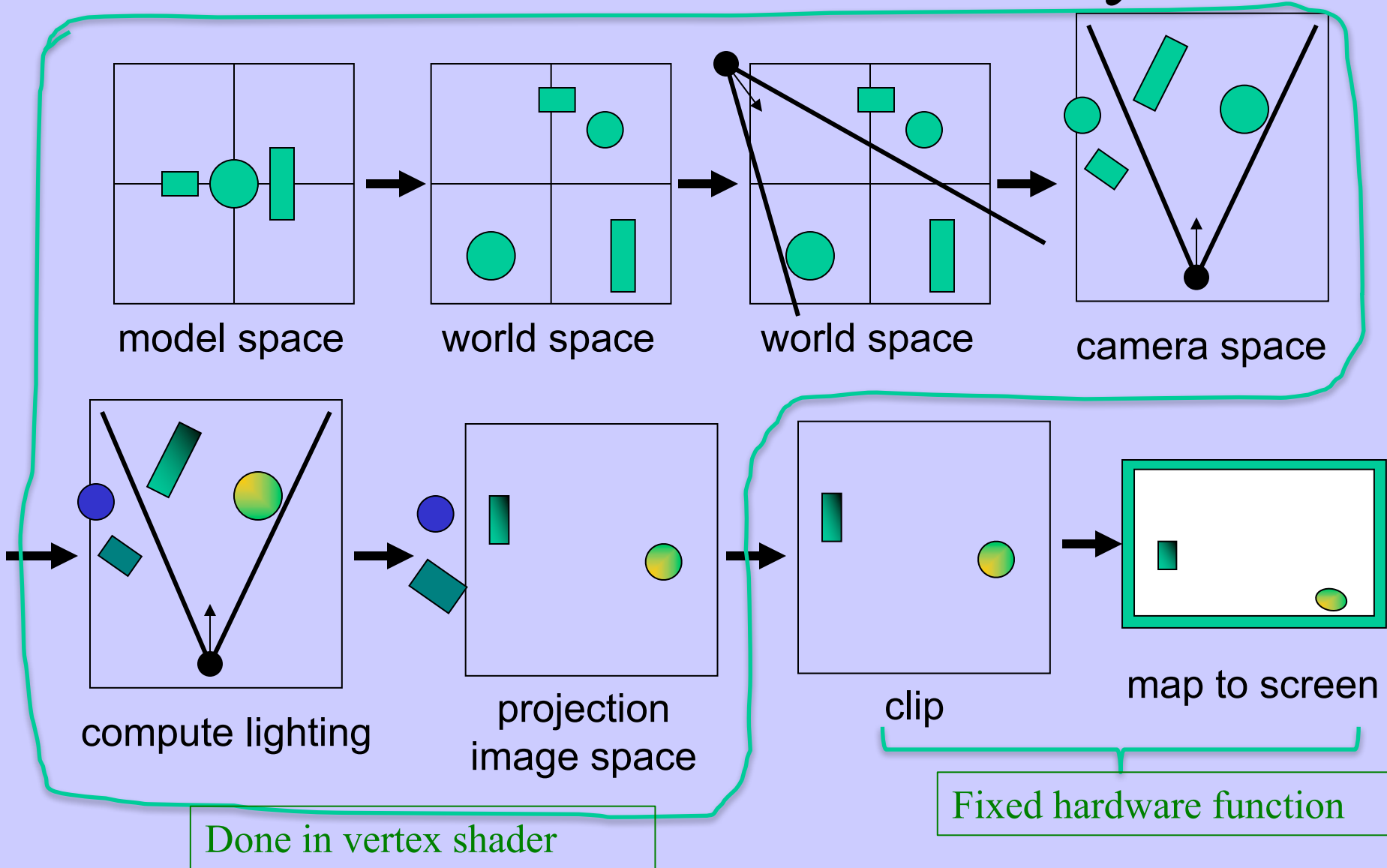
Rewind!
Let's take a closer look

- 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
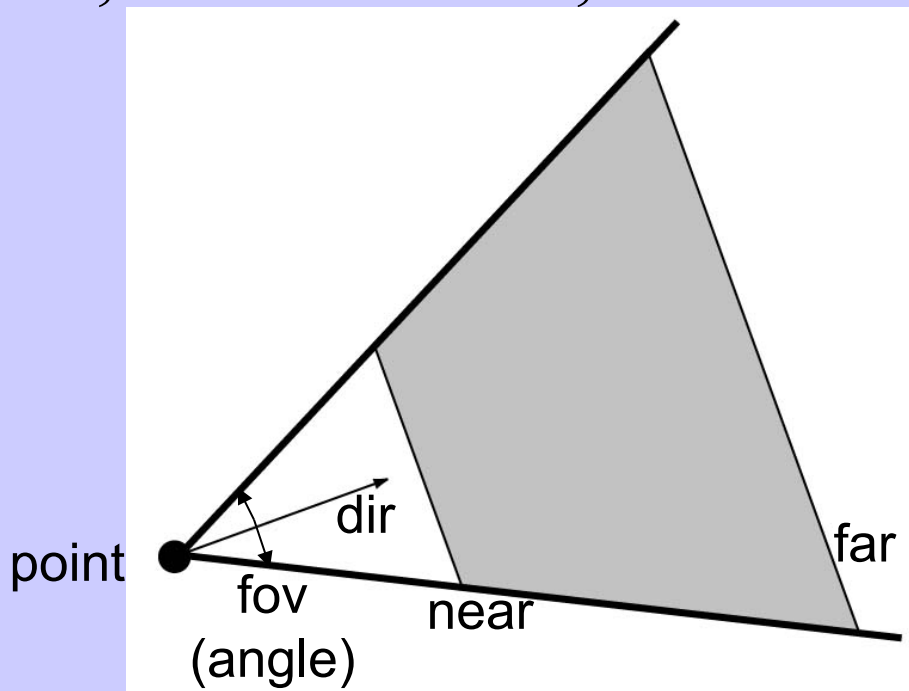
Application → Geometry → Rasterizer

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
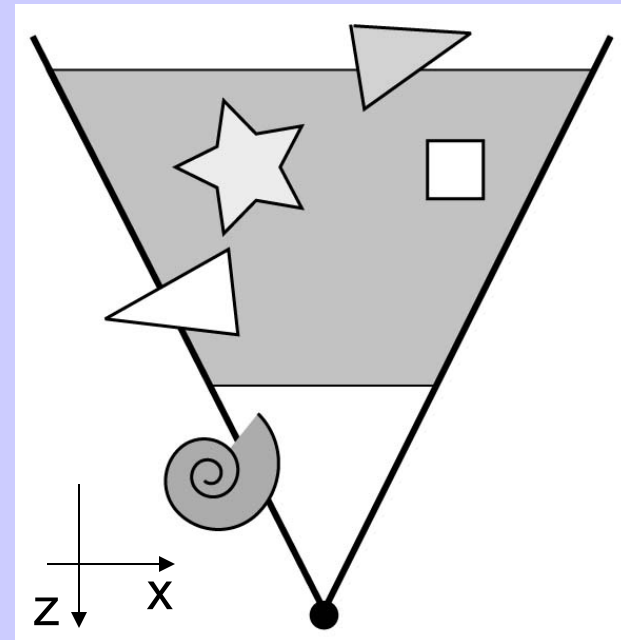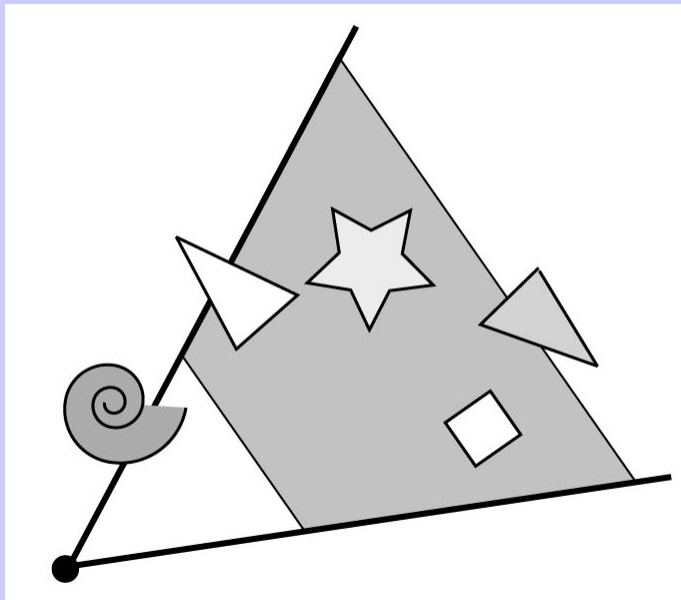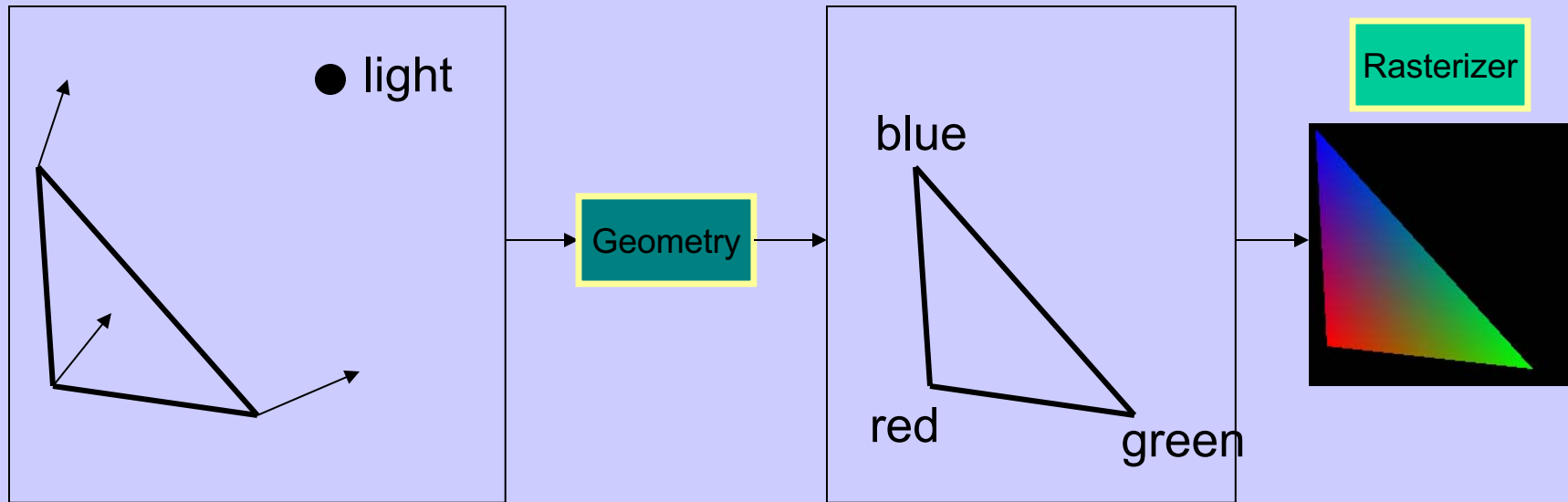
GEOMETRY - Lighting

- Compute "lighting" at vertices

● light

Geometry

blue

red

green
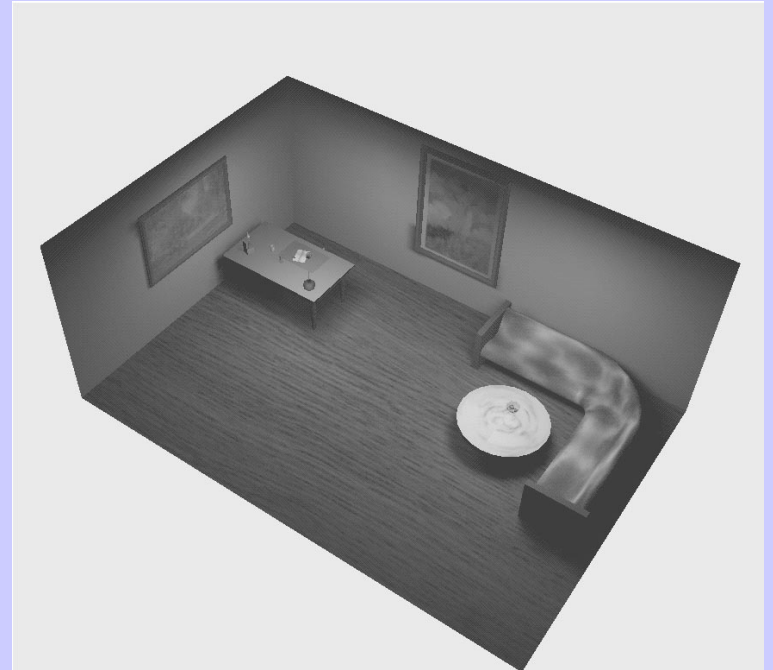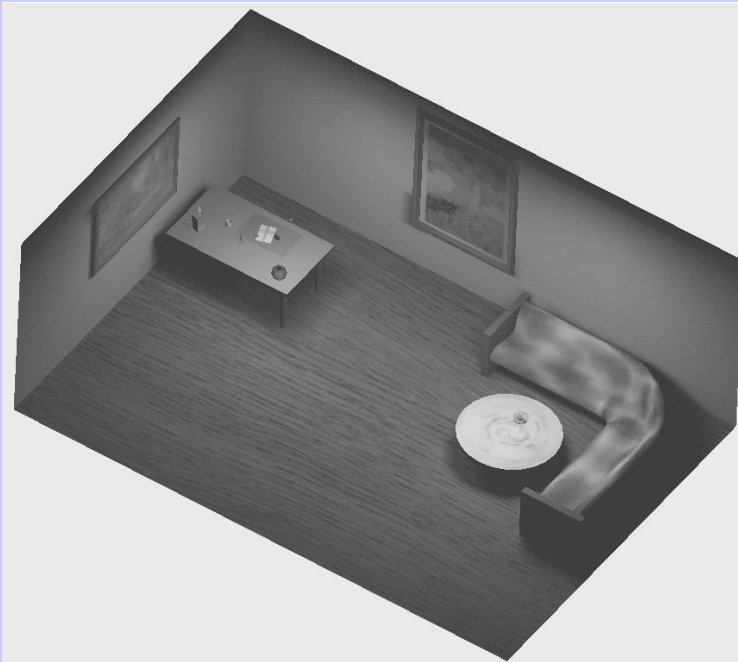
Rasterizer

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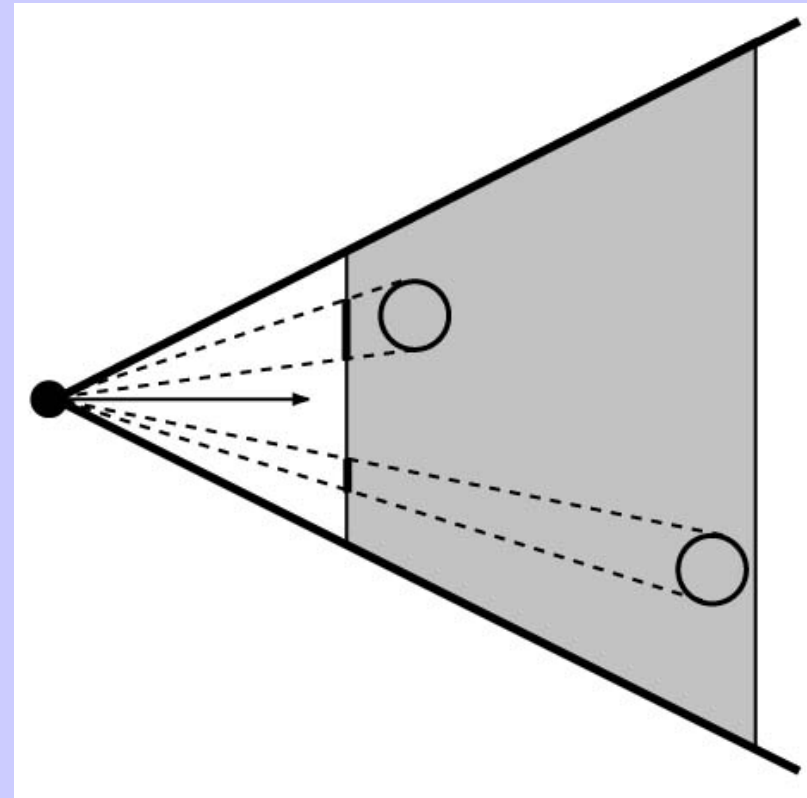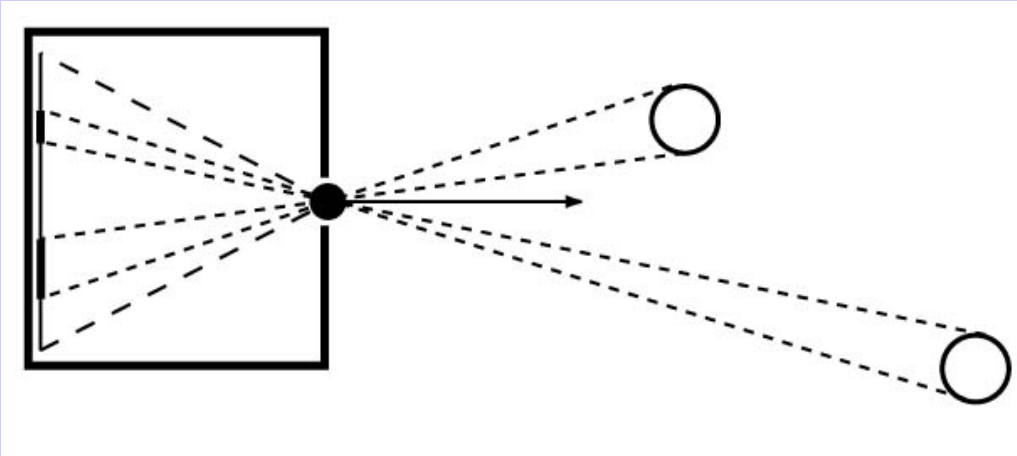- Much more about this in later lecture

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!
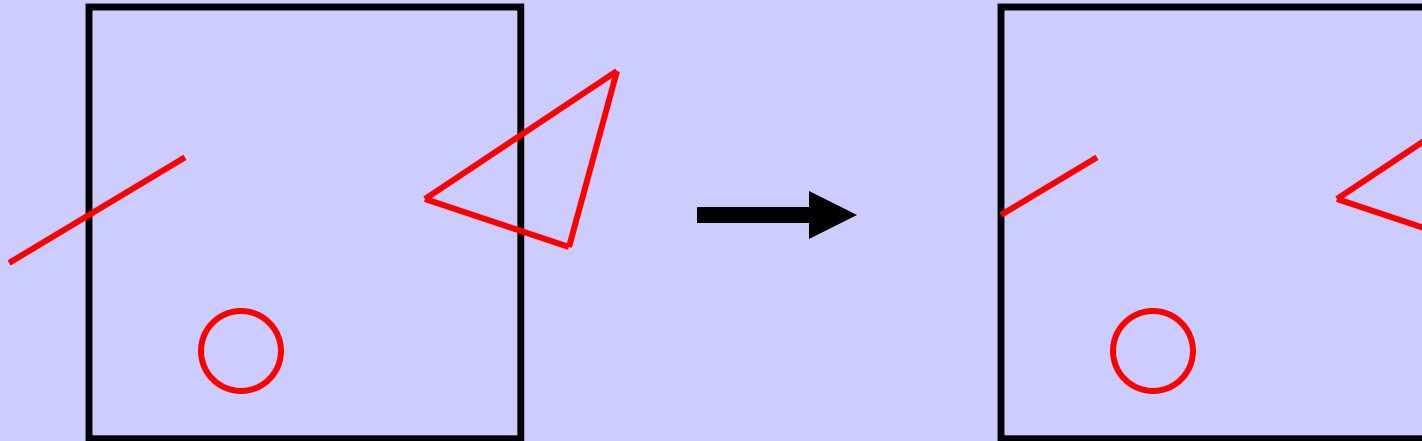- Pinhole camera (left), analog used in CG (right)

GEOMETRY

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
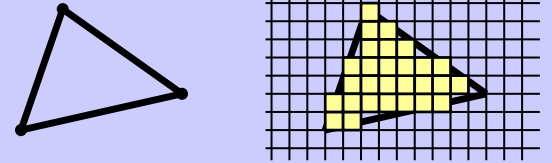- These "screen space coordinates" together with Z (depth) are sent to the rasterizer stage

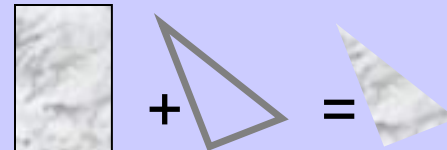# The RASTERIZER in more detail

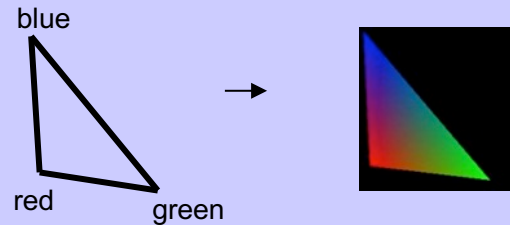- ## Scan-conversion
  - Find out which pixels are inside the primitive

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

blue

red   green

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

- ## Doublebuffering

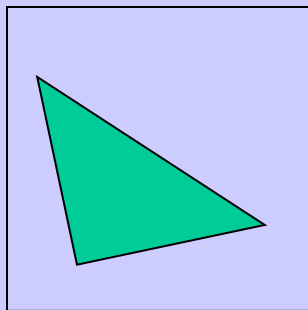# The RASTERIZER
Z-buffering

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

- Assume two equally large tris at different depths

Triangle 1 Triangle 2

incorrect correct
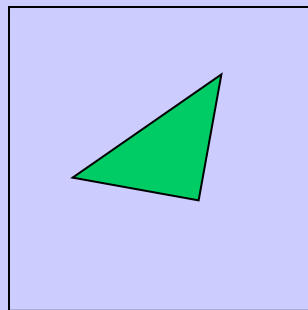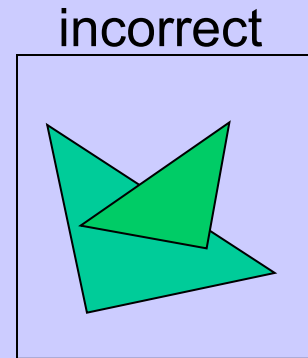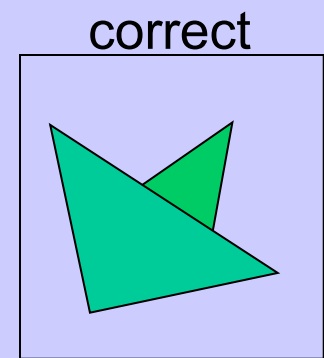
Draw 1 then 2 Draw 2 then 1

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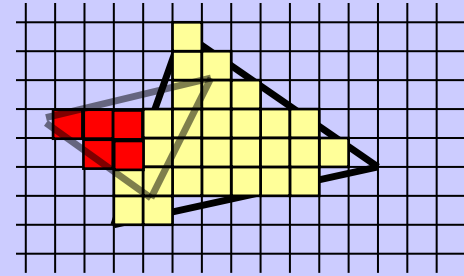
- Would be nice to avoid sorting…
- The Z-buffer (aka depth buffer) solves this
- Idea:
  - Store z (depth) at each pixel
  - When rasterizing a triangle, compute z at each pixel on triangle
  - Compare triangle's z to Z-buffer z-value
  - If triangle's z is smaller, then replace Z-buffer and color buffer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

The RASTERIZER
Z-buffer
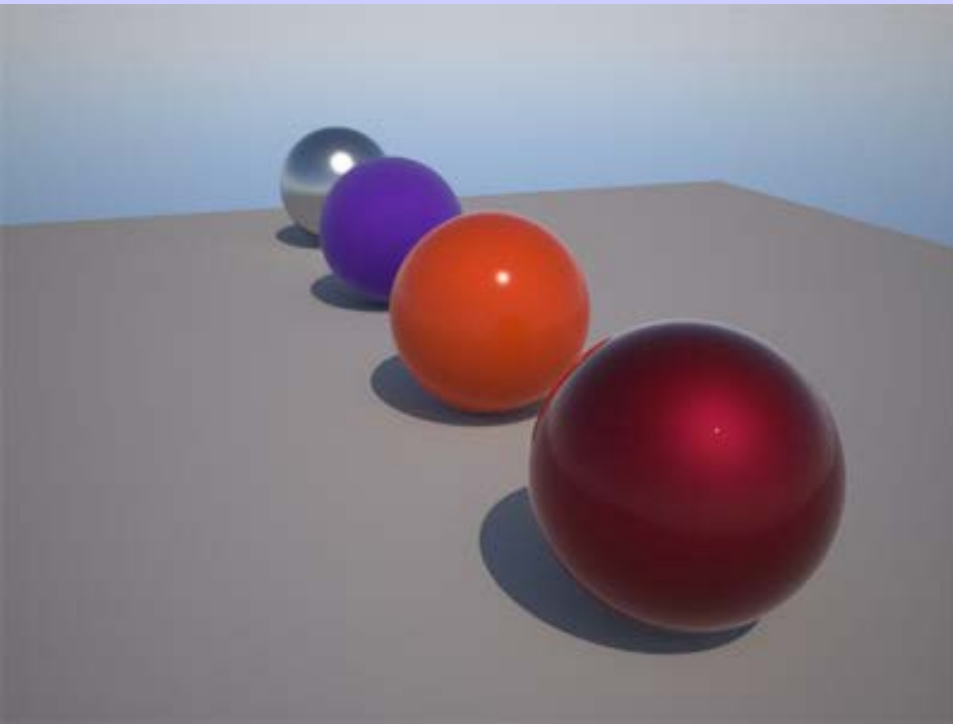
The color buffer
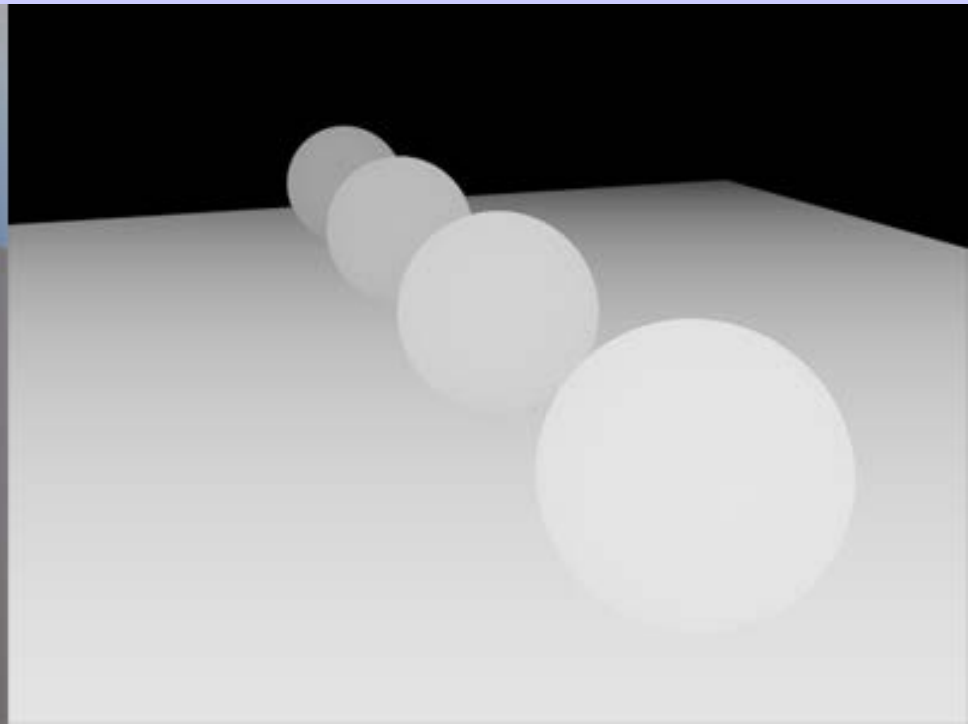
The z-buffer
(or depth buffer)

The RASTERIZER

Z-buffer

Painter's Algorithm

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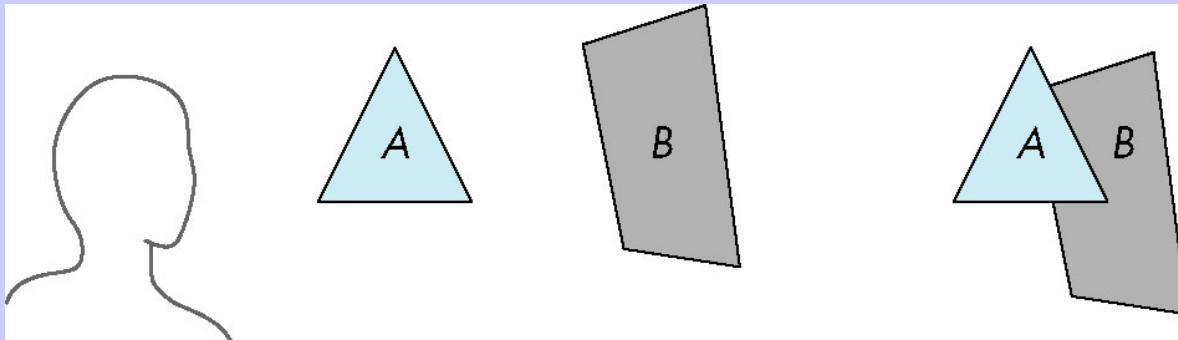
B behind A as seen by viewer

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 (closer to cam), place shade of pixel in color buffer and update z buffer.
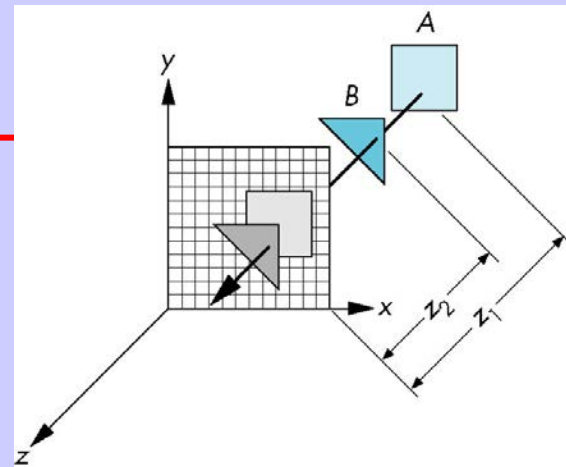
The RASTERIZER double-buffering

- 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 is solved by V-Sync…

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.

What is important:

- Understand the Application-, Geometry- and Rasterization Stage

- Correlation to hardware

- Z-buffering, double buffering, screen tearing

Simple Application...

OLD WAY

OpenGL 1.1

```
#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.
}
```

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();
}
```